"十二五"普通高等教育本科国家级规划教材

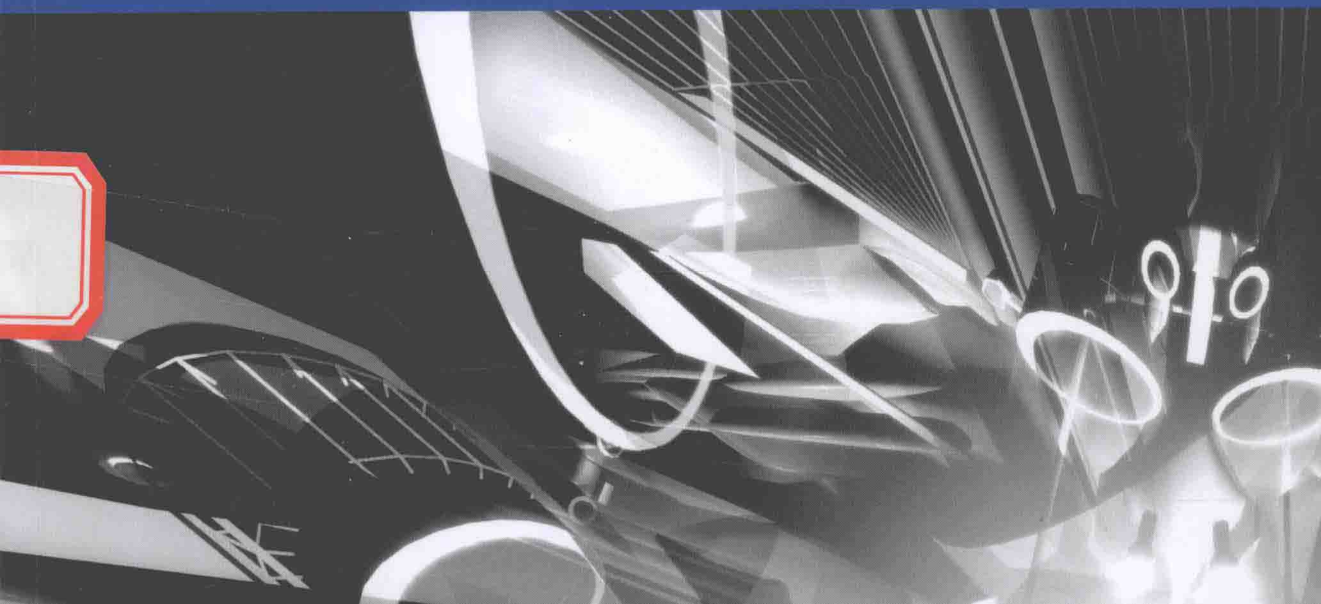
ELECTRONIC
INFORMATION SPECIALTY

Verilog HDL数字集成电路设计原理与应用

（第二版）

蔡觉平　李振荣
何小川　李逍楠　翁静纯　编著

西安电子科技大学出版社
http://www.xduph.com



VISIT…

LANZAROTE Caliente.com

XDUP 440200

封面设计：佳易传播 WWW.SXJYCB.COM



ISBN 978-7-5606-4110-2

定价：35.00 元

"十二五"普通高等教育本科国家级规划教材

Verilog HDL 数字集成电路设计原理与应用

(第二版)

蔡觉平　李振荣　何小川　李逍楠　翁静纯　编著

安徽建筑大学 藏書 图书馆

西安电子科技大学出版社

内 容 简 介

本书系统地对 Verilog HDL 语法和程序设计进行了介绍，明确了数字可综合逻辑设计和测试仿真程序设计在 Verilog HDL 中的不同，通过对典型的组合逻辑电路、时序逻辑电路和测试程序的设计举例，较为完整地说明了 Verilog HDL 在数字集成电路中的使用方法。

全书共 8 章，主要内容包括硬件描述语言和 Verilog HDL 概述，Verilog HDL 的基本语法，Verilog HDL 程序设计语句和描述方式，组合电路和时序电路的设计举例，Verilog HDL 集成电路测试程序和测试方法，较为复杂的数字电路和系统的设计举例，数字集成电路中 Verilog HDL 的 EDA 工具和使用，以及对 Verilog HDL 发展的分析等。

本书可作为电子信息类相关专业本科生和研究生的教材，也可作为数字集成电路设计工程师的参考书。

图书在版编目(CIP)数据

Verilog HDL 数字集成电路设计原理与应用/蔡觉平等编著. —2 版.
—西安：西安电子科技大学出版社，2016.8
“十二五”普通高等教育本科国家级规划教材
ISBN 978 - 7 - 5606 - 4110 - 2

Ⅰ. ① V… Ⅱ. ① 蔡… Ⅲ. ① 数字集成电路—电路设计—高等学校—教材 ② VHDL 语言—程序设计—高等学校—教材 Ⅳ. ① TN431.2 ② TP312

中国版本图书馆 CIP 数据核字(2016)第 125439 号

策划编辑 云立实
责任编辑 买永莲
出版发行 西安电子科技大学出版社(西安市太白南路 2 号)
电　　话 (029)88242885 88201467 邮　　编 710071
网　　址 www.xduph.com 电子邮箱 xdupfxb001@163.com
经　　销 新华书店
印刷单位 虎彩印艺股份有限公司
版　　次 2016 年 8 月第 2 版 2016 年 8 月第 2 次印刷
开　　本 787 毫米×1092 毫米 1/16 印 张 18.5
字　　数 438 千字
定　　价 35.00 元
ISBN 978 - 7 - 5606 - 4110 - 2/TN

XDUP 4402002-2

* * * 如有印装问题可调换 * * *

前　言

随着集成电路技术的飞速发展，集成电路的制造工艺已经达到 14 nm 甚至更小尺寸，数字集成电路的规模越来越大，复杂度越来越高。为了提高设计效率和可靠性，融合了电子技术、计算机技术和智能化技术的 EDA(Electronics Design Automation)工具已经在高速复杂数字集成电路中得到了广泛应用。

硬件描述语言(HDL)是现代专用集成电路(ASIC)的重要设计和仿真语言。目前，大部分数字集成电路设计者都在使用 HDL 的设计方法创建高层次、结构化、基于语言的抽象电路描述，利用已有的设计技术综合出所需硬件电路，并对其进行功能验证和时序分析。

对于准备从事集成电路设计和 FPGA 设计的读者来说，必须了解如何在设计流程的关键阶段正确使用 HDL，并在综合后获得期望的电路，因此需要在 HDL 基本语法结构的基础上，深入理解电路的综合特性和测试仿真方法。本书就是为这样一个目标而撰写的。

Verilog HDL 是被广泛采用的一种硬件描述语言，目前有关 Verilog HDL 的书籍多数都将重点放在了语言和语法的讲解上，而较少分析 Verilog HDL 和相应数字电路的关系，以及如何通过设计得到与目标相符合的电路系统。本书的着眼点主要放在 Verilog HDL 的设计方法上，这也是本书编写的基本出发点。

本书主要根据 Verilog HDL 国际标准 IEEE 1364—2001，对基本语法、设计方法、综合方法和验证方法等进行讲解和分析，并通过大量的实用电路的设计，使读者能够对基于 Verilog HDL 的数字集成电路设计技术有一个全面的了解。

本书重点集中于如何在数字电路设计中的设计、综合和验证过程中合理使用 Verilog HDL。由于 Verilog HDL 本质上是对数字电路的一种抽象描述，因此学习本书时需具备较深入的数字电路设计基础知识，同时至少应熟悉一种编程语言，以助于通过阅读获取有用知识，并提高设计能力。在本书中，我们仅讨论了 Verilog HDL 的核心设计和验证方法，并采用典型的设计例程，以帮助读者快速掌握相关知识。由于数字电路的多样性和灵活性，因此希望在这些例程的基础上，能够为复杂电路的设计提供帮助。

在数字电路中通常采用真值表、状态转移图和算法状态图对组合电路与时序电路进行分析和表示，本书即采用了这些方法用于 Verilog HDL 的设计和分析，以提高读者对设计方法的理解。同时，对于目前在信号处理、自动控制、数值计算等应用中所采用的一些设计方法(如查找表(LUT)、级数展开和有限状态机)进行了说明和举例，希望能够帮助读者扩展设计思路。

本书列举大量实例的主要目的是希望读者在使用 Verilog HDL 的超大规模集成电路(VLSI)设计方法中，学习如何应用关键步骤进行设计和验证。书中所列举的实例都是完整的，并在 ModelSim 和 Synplify 软件中进行了编译、综合和仿真。为了适应 FPGA 设计初学者的需要，本书以 Altera 公司的 FPGA 和工具软件 Quartus 为例，对 FPGA 开发流程进行了讲解。

本书是在第一版(西安电子科技大学出版社，2011 年)的基础上修订而成的，是“十二五”普通高等教育本科国家级规划教材，适用于学习集成电路设计和硬件描述语言课程的本科

生和研究生，可作为课程教材和设计参考。对于希望通过实例学习 Verilog HDL，并将这种语言应用于集成电路设计和测试的专业工程师，本书也会有一定的帮助。本书假定读者已具有布尔代数和数字逻辑设计等背景知识，并具有一定的数字电路设计经验。

全书共分为 8 章。第 1 章主要介绍了 Verilog HDL 在数字集成电路设计中的作用、发展演变的过程和相关国际标准；第 2 章介绍了 Verilog HDL 基本语言结构和语法规则；第 3 章通过例程对 Verilog HDL 设计描述方式和高级程序语句进行了介绍；第 4 章对组合电路和时序电路的 Verilog HDL 程序设计进行了深入分析，并给出了典型的设计例程；第 5 章介绍了 Verilog HDL 仿真测试程序和相关系统任务的知识；第 6 章给出了功能较为复杂的数字电路、系统和总线设计；第 7 章简要介绍了采用 Verilog HDL 进行集成电路设计的 EDA 工具和例程；第 8 章从集成电路设计面临的问题出发，分析了 Verilog HDL 的完善和发展方向。

十分感谢为本书的出版作出贡献的老师和学生。感谢李振荣教授、何小川同学和李涛同学在集成电路设计流程过程、代码质量评估等方面的大量实际工作；感谢李逍楠和张泽同学对于本书中实例的整理和在 ModelSim 与 Synplify 中所做的大量验证工作；感谢翁静纯、张超越、耿剑波同学对于已有著作中语法的重新整理，并提出了在教学中应用的建议；感谢骏龙科技的宋士权先生和程旭先生，以及 Altera 中国的陈卫忠先生长期的支持和帮助。

此外，感谢我的父母和妻子王鸥女士在本书编写和出版过程中对我的支持和鼓励，同时，将本书作为珍贵的礼物送给我的女儿蔡云帆，希望她和我的学生在生活和学习中能够饮水思源、力学笃行。

希望本书能够为致力于集成电路设计的读者和工程师提供帮助。

编　者
2016 年 4 月

目　录

第 1 章　Verilog HDL 数字集成电路设计方法概述 1

1.1　数字集成电路的发展和设计方法的演变 1

1.2　硬件描述语言 3

1.3　Verilog HDL 的发展和国际标准 3

1.4　Verilog HDL 和 VHDL 5

1.5　Verilog HDL 在数字集成电路设计中的优点 6

1.6　功能模块的可重用性 8

1.7　IP 核和知识产权保护 9

1.8　Verilog HDL 在数字集成电路设计流程中的作用 10

本章小结 11

思考题和习题 11

第 2 章　Verilog HDL 基础知识 12

2.1　Verilog HDL 的语言要素 12

2.1.1　空白符 12

2.1.2　注释符 12

2.1.3　标识符和转义标识符 13

2.1.4　关键字 13

2.1.5　数值 14

2.2　数据类型 16

2.2.1　物理数据类型 17

2.2.2　连线型和寄存器型数据类型的声明 19

2.2.3　存储器型 20

2.2.4　抽象数据类型 21

2.3　运算符 22

2.3.1　算术运算符 23

2.3.2　关系运算符 24

2.3.3　相等关系运算符 25

2.3.4　逻辑运算符 26

2.3.5　按位运算符 26

2.3.6　归约运算符 27

2.3.7　移位运算符 28

2.3.8　条件运算符 28

2.3.9　连接和复制运算符 29

2.4　模块 30

2.4.1　模块的基本概念 30

2.4.2　端口 31

本章小结 32

思考题和习题 32

第 3 章　Verilog HDL 程序设计语句和描述方式 34

3.1　数据流建模 34

3.2　行为级建模 36

3.2.1　过程语句 37

3.2.2　语句块 40

3.2.3　过程赋值语句 42

3.2.4　过程连续赋值语句 46

3.2.5　条件分支语句 48

3.2.6　循环语句 52

3.3　结构化建 55

3.3.1　模块级建模 55

3.3.2　门级建模 62

3.3.3　开关级建模 64

本章小结 66

思考题和习题 66

第 4 章　Verilog HDL 数字逻辑电路设计方法 68

4.1　Verilog HDL 的设计思想和可综合特性 68

4.2 组合电路的设计..71
4.2.1 数字加法器..73
4.2.2 数据比较器..76
4.2.3 数据选择器..77
4.2.4 数字编码器..78
4.2.5 数字译码器..83
4.2.6 奇偶校验器..85
4.3 时序电路的设计..86
4.3.1 触发器..92
4.3.2 计数器..94
4.3.3 移位寄存器..95
4.3.4 序列信号发生器..96
4.4 有限同步状态机..100
本章小结..110
思考题和习题..110

第 5 章 仿真验证与 Testbench 编写...113

5.1 Verilog HDL 电路仿真和验证概述........113
5.2 Verilog HDL 测试程序设计基础........114
5.2.1 Testbench 及其结构..114
5.2.2 测试平台举例..117
5.2.3 Verilog HDL 仿真结果确认..120
5.2.4 Verilog HDL 仿真效率..121
5.3 与仿真相关的系统任务..122
5.3.1 $display 和$write..122
5.3.2 $monitor 和 $strobe..124
5.3.3 $time 和 $realtime..127
5.3.4 $finish 和$stop..128
5.3.5 $readmemh 和$readmemb..129
5.3.6 $random..130
5.3.7 值变转储文件系统任务..131
5.4 信号时间赋值语句..135
5.4.1 时间延迟的语法说明..136
5.4.2 时间延迟的描述形式..136
5.4.3 边沿触发事件控制..141
5.4.4 电平敏感事件控制..144
5.5 任务和函数..145
5.5.1 任务..145
5.5.2 函数..149
5.5.3 任务与函数的区别..153
5.6 典型测试向量的设计..153
5.6.1 变量初始化..153
5.6.2 数据信号测试向量的产生..154
5.6.3 时钟信号测试向量的产生..155
5.6.4 总线信号测试向量的产生..157
5.7 用户自定义元件模型..159
5.7.1 UDP 的定义与调用..159
5.7.2 UDP 应用实例..161
5.8 基本门级元件和模块的延时建模..163
5.8.1 门级延时建模..163
5.8.2 模块延时建模..167
5.8.3 与时序检查相关的系统任务..170
5.9 编译预处理语句..171
5.9.1 宏定义..171
5.9.2 文件包含处理..173
5.9.3 仿真时间标度..174
5.9.4 条件编译..175
5.9.5 其它语句..176
5.10 Verilog HDL 测试方法简介..176
本章小结..177
思考题和习题..177

第 6 章 Verilog HDL 高级程序设计举例..181

6.1 数字电路系统设计的层次化描述方法..181
6.2 典型电路设计..185
6.2.1 加法器树乘法器..185
6.2.2 Wallace 树乘法器..188
6.2.3 复数乘法器..191
6.2.4 FIR 滤波器的设计..192
6.2.5 片内存储器的设计..195
6.2.6 FIFO 设计..200
6.2.7 键盘扫描和编码器..204
6.2.8 log 函数的 Verilog HDL 设计..212
6.2.9 CORDIC 算法的 Verilog HDL 实现..215
6.3 总线控制器设计..223

6.3.1 UART 接口控制器 223
6.3.2 SPI 接口控制器 227
本章小结 230
思考题和习题 231

第 7 章 仿真测试工具和综合工具 233
7.1 数字集成电路设计流程简介 233
7.1.1 设计规范 233
7.1.2 设计划分 234
7.1.3 设计输入 234
7.1.4 仿真 234
7.1.5 综合 235
7.1.6 适配布线 235
7.1.7 时序分析 235
7.1.8 物理验证 236
7.1.9 设计结束 236
7.2 测试和仿真工具 236
7.2.1 ModelSim 的使用 237
7.2.2 NC-Verilog 的使用 246
7.3 综合工具 249
7.3.1 Synplify 的使用 249
7.3.2 Design Compiler 的使用 255
7.4 布局布线工具及后仿真 260
7.4.1 工具简介 260
7.4.2 布局布线 262
7.4.3 后仿真 266
7.4.4 添加仿真库 269
7.5 Quartus II 工具全流程应用 271
7.5.1 设计实现 272
7.5.2 仿真验证 275
7.5.3 编程与配置 278
本章小结 281
思考题和习题 281

第 8 章 设计与验证语言的发展趋势 283
本章小结 287
思考题和习题 287

参考文献 288

第 1 章　Verilog HDL 数字集成电路设计方法概述

1.1　数字集成电路的发展和设计方法的演变

从 20 世纪 60 年代开始，数字集成电路的工艺、制造和设计技术飞速发展，数字集成电路从最早的真空管和电子管电路，发展到以硅基半导体为主的集成电路。集成电路的规模从开始的几十个逻辑门的小规模集成电路(Small Scale Integrated，SSI)发展到单芯片数千万个逻辑门的极大规模集成电路(Ultra Large Scale Integrated，ULSI)，单芯片可以集成几十亿只晶体管(见图 1.1-1)。集成电路设计单元从起初的分立元件发展到 IP 复用；系统级别由早期的印刷版系统发展到当下最为流行的片上系统(System on Chip，SoC)；采用的 22 nm 和 16 nm 工艺技术已成熟，并迅速向更小尺寸的产品方向发展；功能方面也从开始的简单布尔逻辑运算发展到可以每秒处理数十亿次计算的复杂运算，使数字集成电路在计算机、通信、图像等领域得到了广泛应用。

集成电路工艺制造水平的提高和芯片规模的扩大，使芯片的设计方法和设计技术发生了很大的变化，如图 1.1-2 所示。早期的数字系统大多采用搭积木式的原理图设计方法，通过一些固定功能的器件加上一定的外围电路构成模块，再由这些模块进一步形成功能电路。这种设计方式的灵活性差，只适合于中小规模的集成电路，当电路和模块的规模增大时，设计效率会降低。

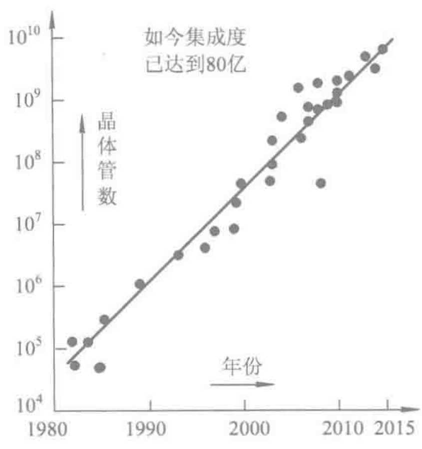


图 1.1-1　数字集成电路复杂度趋势

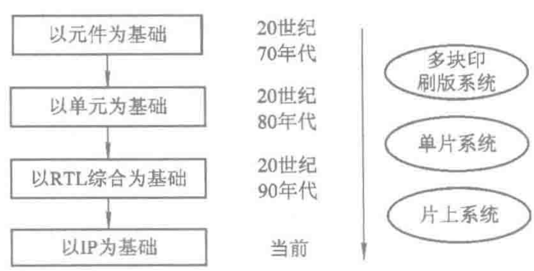


图 1.1-2　数字集成电路设计方法的演变

集成电路的发展可分为三个主要阶段。20 世纪 70 年代(第一次变革时期)，是以加工制造为主导的 IC(Integrated Circuit)产业发展的初级阶段，主流产品是简单微处理器(Micro

Processor Unit，MPU)、存储器以及标准通用逻辑电路。这一时期，IC 整合元件厂(Integrated Device Manufacturer，IDM)在 IC 市场中充当主要角色，设计只作为附属部门而存在。芯片设计和半导体工艺密切相关，设计主要以人工为主，计算机辅助设计(Computer Aided Design, CAD)系统仅作为数据处理和图形编程之用。

20 世纪 80 年代(第二次变革时期)是标准工艺加工线(Foundry)公司与 IC 设计公司共同发展的阶段，主流产品是 MPU、微控制器(Micro Control Unit，MCU)及专用 IC (Application-Specific IC，ASIC)。这时，Foundry 和 IC 设计公司相结合的方式开始成为集成电路产业发展的新模式。这一时期，IC 产业开始进入以客户为导向的阶段。首先，标准化功能的 IC 已难以满足整机客户对系统成本、可靠性等的要求；其次，由于小尺寸加工技术的进步，软件的硬件化已成为可能，超大规模集成电路(Very Large Scale Integrated，VLSI)开始成为主流芯片；再次，随着电子设计自动化(Electronic Design Automation，EDA)工具软件的发展，采用了元件库、工艺模拟参数及其仿真概念等方法，芯片设计开始进入以计算机为主的抽象化软件阶段，使设计过程可以独立于生产工艺而存在。无生产线的 IC 设计公司(Fabless)和设计部门纷纷建立起来并得到迅速的发展，同时以制造为主的 Foundry 工厂也迅速发展起来。1987 年，全球第一个 Foundry 工厂——台湾积体电路公司成立，它的创始人张忠谋被誉为“芯片加工之父”。

20 世纪 90 年代(第三次变革时期)，IC 产业的“四业”(设计业、制造业、封装业、测试业)开始分离，功能强大的通用型中央处理器(Central Processing Unit，CPU)和信号处理器(Digital Signal Processing，DSP)成为产业新的增长点。在这个阶段，芯片厂商认识到，越来越庞大的集成电路产业体系并不利于整个 IC 产业的发展，“分”才能精，“整合”才成优势。于是，IC 产业结构向高度专业化转化成为一种趋势，开始形成设计业、制造业、封装业、测试业独立成行的局面，全球 IC 产业的发展越来越显示出这种结构的优势。

进入 21 世纪，IC 产业的发展速度更是惊人，基于市场和社会发展的需要，数字集成电路正向多元化方向发展。在芯片的市场需求方面，移动通信、多媒体技术等应用的迅速发展，使具有特定功能的差异化专用芯片取代通用型芯片，逐渐成为数字 IC 的主要增长点。在技术方面，出现了新的发展方向。首先，CMOS 模拟技术的发展使得数模混合单芯片集成技术迅速发展，在设计和成本方面体现了巨大优势；其次，应用需求使得存储器在 USLI 芯片中的作用越来越明显，高密度存储器及其 SoC 设计成为设计的热点；再次，单芯片规模的扩大使得单纯依靠提升频率的发展路线出现技术瓶颈，大规模多内核处理器结构成为通用型芯片和 SoC 芯片的主流设计方式。在设计方法方面，采用功能复用 IP(Intelligent Property)的设计方式，成为 IC 设计和商业化的一种主要方式，极大提高了 ULSI 芯片的设计效率和可扩展性。

随着集成电路规模的迅速扩大和复杂度的不断提高，芯片设计和制造成本不断增加，设计、测试和制造工艺中的环节亦随之增多，相应的设计过程变得越来越复杂，因此，设计者希望通过某种手段提高数字集成电路的设计、验证的效率和可靠性。

集成电路单元从起初的分立元件到单元，然后到寄存器传输级，再到 IP 复用技术；系统级别由原先的印刷版系统到当下最为流行的 SoC 片上系统。由图 1.1-1 可以看出，数字集成电路技术的发展速度基本符合摩尔定律，芯片上晶体管的集成数目以每三年翻两番的速度增长着。

超大规模集成电路的发展给设计者和开发者提出了一系列问题，如高层次综合、数模混合电路描述、仿真验证与形式验证等自动验证手段、数字电路的超深亚微米效应，以及设计重用等。这些问题给 EDA(Electronic Design Automation，电子设计自动化)技术的发展提出了一系列新的课题。为了从更高的抽象层次开展设计工作，增强元件模型的可重用性，提高硬件描述设计效率，提出了采用硬件描述语言(Hardware Description Language，HDL)进行数字集成电路设计。如何自动化、高效率地进行数字电路的设计，是 HDL 产生的出发点，也是其进一步完善和发展的目标。

1.2 硬件描述语言

C、FORTRAN、Pascal 等程序化设计语言，极大地提高了计算机软件程序设计的效率和可靠性。因此，在硬件设计领域，设计人员也希望采用程序化设计语言来进行硬件电路的设计。为此，产生了硬件描述语言 HDL。HDL 是一种高级程序设计语言，通过对数字电路和系统的语言描述，可以对数字集成电路进行设计和验证。利用 HDL，数字集成电路设计工程师可以根据电路结构的特点，采用层次化的设计结构，将抽象的逻辑功能用电路的方式进行实现。为了提高 HDL 对数字电路设计、综合和仿真的能力，Mentor、Cadence、Synopsys 等公司提供了功能强大的电子设计自动化(EDA)工具，可以将 HDL 程序综合成为网表，通过自动布局布线工具把网表转换为具体电路布线结构，用于专用集成电路(Application Specific Integrated Circuit，ASIC)和现场可编程门阵列(Field Programmable Gate Array，FPGA)的实现。

HDL 发展至今，产生了很多种对于数字集成电路的描述性设计语言，并成功地应用于设计的各个阶段(建模、仿真、验证和综合等)。20 世纪 80 年代至今，已出现了上百种硬件描述语言，它们对设计自动化起到了极大的促进和推动作用，主要有 Gateway Design Automation 公司提出的 Verilog HDL、美国国防部高级研究计划局(DARPA)设计的 VHDL、美国国防部 RPASSP(Rapid Prototyping of Application Specification Signal Processing)计划提出的基于面向对象的 OO VHDL(Object Oriented VHDL)、美国杜克大学的 DE VHDL(Duke Extended VHDL)和美国电气和电子工程师协会(Institute of Electrical and Electronics Engineers，IEEE)支持的 VITAL 等。Verilog HDL 和 VHDL 是目前主要的两种 HDL 语言，并分别在 1995 年和 1987 年被采纳为 IEEE 国际标准，广泛用于数字集成电路的设计和验证领域。

1.3 Verilog HDL 的发展和国际标准

Verilog HDL 是一种常用的硬件描述语言，可以从系统级、电路级、门级到开关级等抽象层次进行数字电路系统的建模、设计和验证工作。利用该语言可以设计出简单的门级电路，甚至功能完整的数字电路系统。

从 Verilog HDL 的设计之初到目前的广泛应用，经过 30 多年的发展，其功能也由最初的模拟集成电路设计发展到数字和模拟电路设计(见图 1.3-1)，它已经成为数字电路和数字

集成电路中使用最为广泛的设计语言。

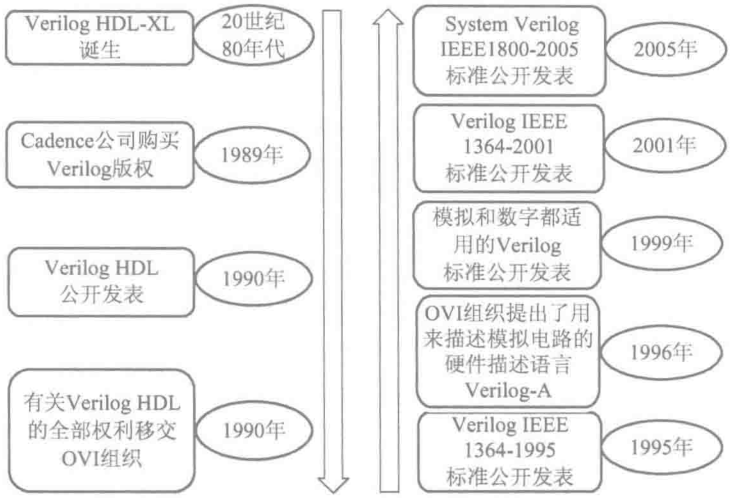


图 1.3-1 Verilog HDL 的发展历史

Verilog HDL 最初是由 Gateway Design Automation(GDA)公司于 1983 年为其模拟器产品开发的硬件建模语言。作为一种便于使用的专用设计语言，Verilog HDL 被广泛用于模拟集成电路和仿真器中，并逐渐为众多设计者所接受。在随后的几年，Verilog HDL 开始在数字电路设计领域广泛使用。1987 年，Synopsys 公司开始使用 Verilog HDL 作为综合工具的输入，为在数字集成电路上的应用提供了 EDA 综合工具，提高了电路描述性设计方式的效率。

1989 年 Cadence 公司收购了 GDA 公司，Verilog HDL 成为 Cadence 公司的专有设计语言。为了在更大范围内推广和使用 Verilog HDL，1990 年 Cadence 公司决定公开 Verilog HDL，于是成立了 OVI(Open Verilog International)组织，负责促进 Verilog HDL 的发展。

1993 年，几乎所有 ASIC 厂商都开始支持 Verilog HDL，并且认为 Verilog HDL-XL 是最好的仿真器。同时，OVI 组织推出 Verilog HDL 2.0 规范，IEEE 接受了将其作为 IEEE 标准的提案。自此，Verilog HDL 正式成为数字集成电路的设计语言标准(见表 1.3-1)。

表 1.3-1 Verilog HDL 国际标准

名 称	时 间	备 注
Verilog IEEE 1364-1995	1995 年 12 月	基于 Verilog HDL 的优越性，IEEE 制定了 Verilog HDL 的 IEEE 标准，即 Verilog HDL 1364-1995
Verilog-A	1996 年	Verilog-A 是由 OVI 组织提出的一种硬件描述语言，是模拟电路行业的标准建模语言，来源于 IEEE 1364 Verilog 规范
	1999 年	模拟和数字都适用的 Verilog 标准公开发布
Verilog IEEE 1364-2001	2001 年	IEEE 制定了 Verilog IEEE 1364-2001 标准，并公开发布；其中 HDL 部分相对于 1995 年的标准有较大增强
System Verilog IEEE 1800-2005	2005 年	此标准是继 VHDL 和 Verilog HDL 之后仿真工具支持的语言，它建立在 Verilog HDL 的基础上，是 IEEE 1364 Verilog-2001 标准的扩展，兼容 Verilog-2001，将成为下一代硬件设计和验证的语言

1995 年底，IEEE 制定了第一个 Verilog HDL 标准 Verilog IEEE 1364-1995。在此基础上，

于 2001 年又增加了部分功能，并制定了较为完善的标准 Verilog IEEE 1364-2001。目前在数字集成电路方面主要采用的就是这两个标准所规定的程序语法和设计规范。

Verilog HDL 在数字集成电路设计上的优越性，使其在硬件设计领域得到了广泛的应用和发展。

在模拟电路设计方面，基于 IEEE 1364 Verilog HDL 规范，提出了模拟电路行业的标准建模语言 Verilog-A，以提高模拟集成电路的程序化设计能力。

在系统级设计方面，传统的设计方法采用 C 语言等高级软件语言进行数学模型的建立和分析，通过定点化设计，将数学模型转变成电路模型，最后采用 HDL 进行电路设计。这种方法的缺点是，数学模型的建立和电路设计是独立的，从而导致设计周期长、需要的人员和软件多，且存在重复性的工作等问题。研究和开发人员希望能将数学模型直接用于数字集成电路的设计，以提高集成电路的设计效率，这就给 EDA 工具厂商提出了新的要求。为了满足这一要求，2005 年诞生了 System Verilog IEEE 1800-2005 标准。该标准建立在 Verilog HDL 的基础上，在系统层次上增强了模型建立和验证的功能，是 Verilog IEEE 1364-2001 标准的扩展，向下兼容 Verilog-2001，并将成为下一代硬件设计和验证的语言。

在功能设计方面，Verilog HDL 采用描述性建模方式，通过行为描述、数据流描述和结构性描述等方式，可以对电路、输入信号激励和响应监控方式进行设计。同时，提供编程语言接口，通过该接口可以在模拟、验证期间从设计外部访问设计，包括模拟的具体控制和运行。

Verilog HDL 定义了完善的语法规则，对每个语法结构都定义了清晰的模拟、仿真语义。它从 C 语言中继承了多种操作符和结构，具有较强的扩展建模能力。Verilog HDL 的核心子集相对紧凑，可以满足大多数建模应用的要求，容易学习和掌握。当然，应用于数字集成电路设计的较为完整的 Verilog HDL 还有很多的语法规则和使用方式，需要进一步学习。本书主要针对 Verilog HDL 基本语法规则和数字集成电路设计进行介绍，更为专业和细致的内容需要参照相关的国际标准和 EDA 工具的功能说明，以应对越来越复杂的数字集成电路芯片设计和验证工作。

1.4　Verilog HDL 和 VHDL

目前，最为常用的硬件描述语言有两种，分别是 Verilog HDL 和 VHDL(VHSIC Hardware Description Language)。其中，VHSIC 是 Very High Speed Integrated Circuit 的缩写，故 VHDL 准确的中文译名应为甚高速集成电路的硬件描述语言。

Verilog HDL 和 VHDL 都是完备的 HDL 设计和验证语言，具有完整的设计方法和设计规范。它们可以设计和验证超大规模数字集成电路，并且都已成为 IEEE 标准。选用哪种语言进行数字集成电路开发，主要取决于设计单位的基础、计划采用的设计方案和 EDA 工具。这两种 HDL 具有较多的共同特点：

(1) 能形式化地抽象表示电路的行为和结构；

(2) 支持逻辑设计中层次与范围的描述；

(3) 可借用高级语言的精巧结构来简化电路行为的描述，具有电路仿真与验证机制，以保证设计的正确性；

(4) 支持电路描述由高层到底层的综合转换；

(5) 硬件描述与实现工艺无关(有关工艺参数可通过语言提供的属性包括进去)；

(6) 便于文档管理；

(7) 易于理解和设计重用。

但是作为两种不同的标准化 HDL，Verilog HDL 和 VHDL 在设计方法和范围方面也有一些各自的特点。

(1) 在设计方法方面，VHDL 语法结构紧凑、灵活性差、设计规则繁琐，初学者需要用较长时间掌握它。由于语法规则严谨性高，VHDL 可综合性和代码一致性很强，适用于规模较大的数字集成电路系统设计。而 Verilog HDL 的语法结构和设计方式灵活，初学者对语言掌握的难度较小，设计也较容易进行综合和验证。但是由于设计代码风格的多样性，当数字电路规模较大时，代码的管理和系统设计难度较大。当然，作为经验丰富的数字电路设计工程师，采用何种语言进行设计的关键在于对语言和电路的掌握能力和对设计规范的理解程度。

为了发挥两种设计语言在设计方面各自的优势，EDA 工具厂商提供了 Verilog HDL 和 VHDL 的混合设计、验证和综合方法。因此，设计人员只需掌握其中一种 HDL 即可。

(2) 在设计范围方面，Verilog HDL 和 VHDL 有一个显著的区别：Verilog HDL 可以描述系统级(System)、算法级(Algorithm)、寄存器传输级(RTL)、门级(Gate)和开关级(Switch)电路，而 VHDL 则不具备开关级电路描述能力。在 FPGA 和 CPLD 等用户可配置数字电路的设计中，由于最小可配置电路是门级电路，没有开关级可配置电路，因此两种语言的设计能力相当。但是在专用数字集成电路设计和开关级描述方面，Verilog HDL 的设计范围比 VHDL 略大一些。

图 1.4-1 是 Verilog HDL 和 VHDL 在电路建模能力方面的比较。随着数字集成电路工艺和设计方法的快速发展，这两种设计语言也在不断丰富和改进，以满足更大、更高速、更复杂的数字集成电路系统设计的要求。

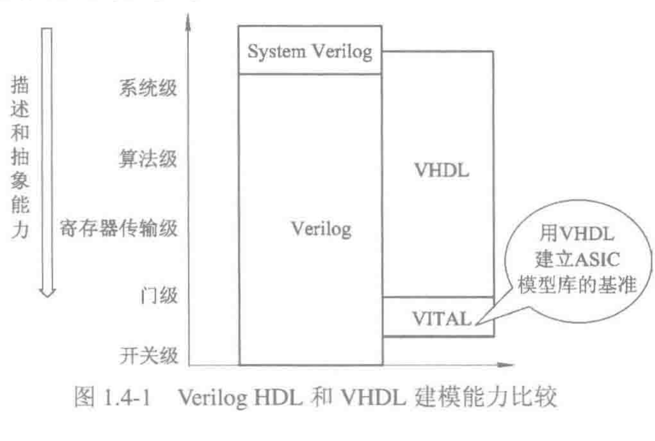


图 1.4-1 Verilog HDL 和 VHDL 建模能力比较

1.5 Verilog HDL 在数字集成电路设计中的优点

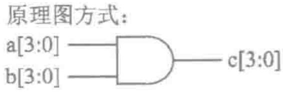
在数字集成电路出现的最初几十年中，数字逻辑电路和系统的设计规模较小，复杂度

也低。ASIC、FPGA 和 CPLD 的设计工作采用厂家提供的专用电路图工具，通过连接线将定制电路单元进行互连实现。随着电路规模的增加，设计人员通常要花费很多的时间做大量重复的手工布线工作，同时为了达到设计目标，对于大量定制单元电路要分厂熟悉。这种低效率的设计方式持续了很长时间。

Verilog HDL 和 EDA 工具的出现和发展，通过运用高效率的描述性语言和强大的仿真综合工具，使设计人员将注意力集中于系统、算法和电路结构上，极大地提高了设计输入和验证的效率。

作为最广泛采用的 HDL，Verilog HDL 在硬件描述方面的效率高、灵活性强。图 1.5-1 中的(a)和(b)分别是 4 位和 32 位总线与逻辑的原理图设计和 Verilog HDL 描述方式的对比。

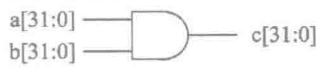
原理图方式：




```
module aand4(a,b,c);
        input [3:0]a,b;
        output [3:0]c;
        reg [3:0] c;
        always @(a or b)
        c=a&b;
endmodule
```

(a) 4位总线与逻辑

原理图方式：



Verilog HDL描述方式：

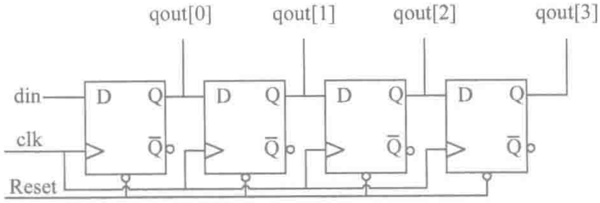
```
module aand32(a,b,c);
        input [31:0]a,b;
        output [31:0]c;
        reg [31:0] c;
        always @(a or b)
        c=a&b;
endmodule
```

(b) 32位总线与逻辑

图 1.5-1 组合逻辑电路原理图设计和 Verilog HDL 描述方式的对比

图 1.5-2 中的(a)、(b)分别是长度为 4 位和 8 位移位寄存器的原理图设计与 Verilog HDL 描述方式的对比。

原理图方式：

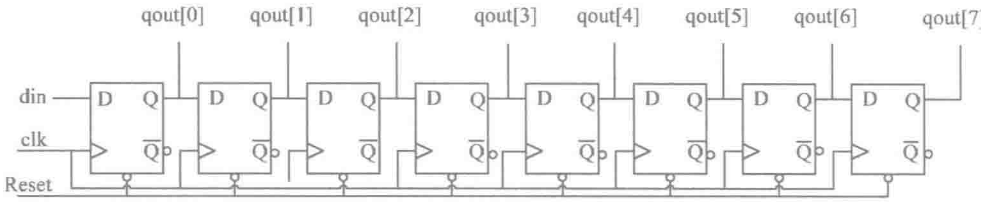


Verilog HDL描述方式：

```
    module shiftregist4(clk,din,Reset, qout);
    input clk,Reset,din;
    output [3:0] qout;
    reg [3:0] qout;
    always @(posedge clk or posedge Reset)
          begin
           if(Reset)  qout<=4'b0000;
           else
                qout[3:0]={qout[2:0], din};
           end
    endmodule
```

(a) 4位移位寄存器

原理图方式：



Verilog HDL描述方式：

```
module shiftregist8(clk,din, Reset,qout);
input clk,Reset,din;
output [7:0] qout;
reg [7:0] qout;
always @(posedge clk or posedge Reset)
   begin
     if(Reset)  qout<=8'b00000000;
     else
            qout[7:0]={qout[6:0], din};
   end
endmodule
```

(b) 8位移位寄存器

图 1.5-2　时序逻辑电路原理图设计和 Verilog HDL 描述方式的对比

图 1.5-1 和图 1.5-2 分别是典型的组合逻辑电路和时序逻辑电路。从这两个例子可以看到，Verilog HDL 在设计方面有两个突出的能力。第一，可以用较少的语句描述较为复杂的电路。图 1.5-1 和图 1.5-2 中采用一条有效语句即实现了电路设计。第二，Verilog HDL 具有极为灵活的可扩展特性。图 1.5-1 中，Verilog HDL 仅需修改总线的位宽，即可将 4 位总线与逻辑转变为 32 位总线与逻辑。图 1.5-2 中仅需改变移位信号的长度，就可以实现不同长度移位寄存器的设计。

通过这两个例子可以看到，Verilog HDL 极大地提高了原理图设计的效率，同时提高了设计的灵活性和电路设计管理的有效性。

1.6　功能模块的可重用性

HDL 的标准化极大地扩展了 Verilog HDL 和 VHDL 的使用范围，并增强了其通用性。目前绝大多数的数字集成电路和 FPGA 的开发采用了 HDL。这使得 Verilog HDL 和 VHDL 的功能模块积累得越来越多，同时也极大地提高了功能模块的可重用性。

由于模块的可重用性对于硬件电路开发效率的提高至关重要，因此业界提出了数字集成电路的软核、硬核和固核的概念。

软核(Soft Core)一般是指经过功能验证、5000 门以上的可综合 Verilog HDL 或 VHDL 模型。软核通常与设计方法和电路所采用的工艺无关，具有很强的可综合性和可重用性。由软核构成的器件称为虚拟器件，通过 EDA 综合工具可以把它与其它数字逻辑电路结合起来，构成新的功能电路。软核的可重用性大大缩短了设计周期，提高了复杂电路的设计能力。

固核(Firm Core)通常是指在 FPGA 器件上，经过综合验证、大于 5000 门的电路网表文件。

硬核(Hard Core)通常是指在 ASIC 器件上，经过验证，正确的、大于 5000 门的电路结构版图掩模。

软核、固核和硬核是目前数字集成电路功能单元模块在不同层级使用的三种形式。由于“软核”采用可读性较高的可综合 HDL 实现，因此其可维护性和可重用性程度高，使用也更加灵活和便捷。固核和硬核是针对不同芯片平台的功能单元，性能稳定，不易修改。商用“软核”通常都有针对不同芯片和工艺而定制的“硬核”和“固核”，可以从不同层次提高数字电路功能模块的可重用性。

目前国际设计领域正试图通过建立相应的标准化组织，推广和规范软核的使用方式，如虚拟接口联盟(Virtual Socket Interface Alliance)希望对接口的标准化来提高 HDL 设计模块的可重用性。

1.7 IP 核和知识产权保护

软核的产生和推广，为集成电路的设计和开发提供了一种新的商业模式，采用 HDL 的可综合代码已成为集成电路和系统产业中的重要产品。现在，超大规模的 ASIC 和 FPGA 设计更多采用的是不同公司功能模块的组合，通过开发特定功能的部件电路，形成具有特定功能的芯片和系统。相应的内核成为各个公司重要的资产，并拥有特殊的知识产权。

IP 核是具有知识产权核的集成电路芯核的总称，是经过反复验证的、具有特定功能的宏模块，且该模块与芯片制造工艺无关，可以移植到不同的半导体工艺中。到了 SoC 阶段，向用户提供 IP 核服务已经成为可编程逻辑器件提供商的重要任务。在 SoC 芯片的设计生产过程中，芯片的生产厂家只需根据设计需要购入相应功能的 IP 核，再将这些 IP 核按照设计要求进行组合，即可完成所需特定功能的设计，如图 1.7-1 所示。这样可以大大减少设计人力的投入并降低风险，缩短设计周期，确保产品质量。

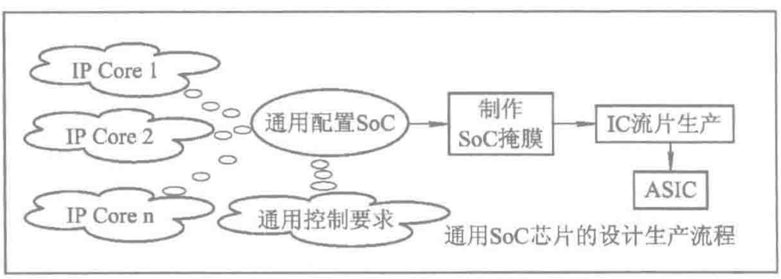


图 1.7-1 采用 IP 核进行开发的 SoC 设计

对于可编程提供商来说，能够提供的 IP 核越丰富，用户的设计就会越方便，其市场占有率就越高。现在，IP 核已经成为系统设计的基本单元，并作为独立设计成果被交换、转让和销售。

目前，全球最大的 IP 设计公司是英国的 ARM 公司，通过 IP 的市场推广，不同性能的 ARM 被广泛用于通信、计算机、媒体控制器、工业芯片中，极大地提高了设计的效率。这种商业模式为集成电路的发展作出了重要贡献。

1.8 Verilog HDL 在数字集成电路设计流程中的作用

图 1.8-1 为一般的数字集成电路设计流程。作为一种标准化的硬件电路设计语言，Verilog HDL 在设计和验证中起着重要作用。

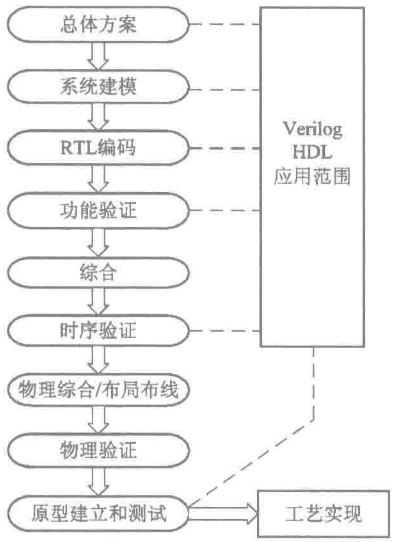


图 1.8-1 数字集成电路设计流程

数字集成电路和 FPGA 设计过程主要分为四个阶段。第一阶段是系统设计阶段，包括总体方案和系统建模两个主要过程。总体方案是对系统进行结构规划、功能分割和互连模型系统级规划的工作。系统建模是对总体方案的细化，将总体方案划分为具体的功能模块，并对互连总线等进行较为详细的设计。

第二阶段是数字电路设计和代码编写阶段，即 RTL 代码编写阶段。在这个阶段，设计人员将系统设计的功能模块进行具体的电路设计，并形成可以测试的功能代码。

第三阶段是电路验证阶段，主要包括对硬件描述语言程序代码的功能验证和经过 EDA 综合工具后的时序验证两个部分。Verilog HDL 程序可以对代码的功能进行基本逻辑的初步验证。Verilog HDL 也可以对程序综合后的生成电路进行时序验证，电路的网表也可以用 Verilog HDL 程序形式表示。

第四阶段是集成电路的后端设计阶段，主要通过 EDA 工具进行物理综合、布局布线、物理验证、原型建立和测试，并最终交付工艺实现。

在集成电路的设计流程中，以 Verilog HDL 为代表的 HDL 发挥了很大作用。在第一、二阶段的电路设计过程中，Verilog HDL 主要进行系统级和电路级的设计和验证；在第三、四阶段，对于不同阶段的综合网表和物理电路的验证工作，Verilog HDL 也被用于电路的验证工作。

因此，Verilog HDL 用于复杂数字逻辑电路和系统的总体仿真、子系统仿真和具体电路

综合等各个设计阶段，在设计流程中具有重要的作用。

本 章 小 结

经过 30 多年的发展和应用，Verilog HDL 已经成为超大规模数字集成电路和 FPGA 等的主要设计语言和设计方法。在设计和验证方面的优越性使其不断完善，极大地提高了数字集成电路的设计能力。

以 HDL 为基础的 IP 技术，进一步提高了设计的效率，并为集成电路产业提供了一种新的合作方式和商业模式。Verilog HDL 正在数字集成电路的设计、验证和综合等方面发挥着越来越重要的作用。Verilog HDL 已经成为数字集成电路设计的重要基础，因此要熟练掌握。

思考题和习题

1. 谈谈你对数字集成电路技术的认识。
2. 什么是硬件描述语言，其主要作用是什么？
3. 当前，符合 IEEE 标准的硬件描述语言有哪两种，各有什么特点？
4. 简述用硬件描述语言进行设计的优、缺点。
5. 为什么说可以用 Verilog HDL 构成非常复杂的电路结构？
6. 为什么可以用比较抽象的描述来设计具体的电路结构？
7. 简述进行 IC 设计的方法和设计流程。
8. 简述 IP 复用技术、软核、硬核、固核的概念。
9. 简述 System Verilog 的特点以及和 Verilog HDL 的异同。
10. 列举常见的 IC 设计、仿真、综合等工具的名称。

第 2 章　Verilog HDL 基础知识

2.1　Verilog HDL 的语言要素

Verilog HDL 语法来源于 C 语言基本语法，其基本词法约定与 C 语言类似。程序的语言要素也称为词法，是由符号、数据类型、运算符和表达式构成的，其中符号包括空白符、注释符、标识符和转义标识符、关键字、数值等。

2.1.1　空白符

空白符包括空格符(\b)、制表符(\t)、换行符和换页符。空白符使代码看起来结构清晰，阅读起来更方便。在编译和综合时，空白符被忽略。

Verilog HDL 程序可以不分行，也可以加入空白符采用多行编写。

例 2.1-1　空白符使用示例。

```
initial begin a = 3'b100; b = 3'b010; end
```

相当于：

```
initial
    begin
        a = 3'b100;
        b = 3'b010;
    end
```

2.1.2　注释符

Verilog HDL 中允许插入注释，标明程序代码功能、修改、版本等信息，以增强程序的可阅读性和帮助管理文档。Verilog HDL 中有两种形式的注释。

(1) 单行注释：单行注释以“//”开始，Verilog HDL 忽略从此处到行尾的内容。

(2) 多行注释：多行注释以“/*”开始，到“*/”结束，Verilog HDL 忽略其中的注释内容。需要注意的是，多行注释不允许嵌套，但是单行注释可以嵌套在多行注释中。

例 2.1-2　注释符使用示例。

单行注释：

```
assign a = b & c;                //单行注释
```

多行注释：

```
assign a[3:0] = b[3:0]&c[3:0;];        /*注释行 1
                   注释行 2 */
```

非法多行注释：

```
/*注释内容 /*多行注释嵌套多行注释*/ 注释内容*/
```

合法多行注释：

```
/*注释内容 //多行注释嵌套单行注释*/
```

2.1.3 标识符和转义标识符

在 Verilog HDL 中，标识符(Identifier)被用来命名信号、模块、参数等，它可以是任意一组字母、数字、$符号和_(下划线)符号的组合。应该注意的是，标识符的字母区分大小写，并且第一个字符必须是字母或者下划线。

例 2.1-3 以下标识符都是合法的。

```
count
COUNT          //与 count 不同
_CC_G5
B25_78
SIX
```

例 2.1-4 以下标识符都是不正确的。

```
30count        //标识符不允许以数字开头
out*           //标识符中不允许包含字符 *
a+b-c          //标识符中不允许包含字符 + 和 -
n@238          //标识符中不允许包含字符@
```

为了使用标识符集合以外的字符或标号，Verilog HDL 规定了转义标识符(Escaped Identifier)。采用转义标识符，可以在一条标识符中包含任何可打印的字符。转义标识符以"\"(反斜线)符号开头，以空白结尾(空白可以是一个空格、一个制表字符或换行符)。

例 2.1-5 以下是合法的转义标识符。

```
\ a+b = c
\7400
\.*.$
\{******}
\~Q
\OutGate    //与 OutGate 相同
```

2.1.4 关键字

Verilog HDL 内部已经使用的词称为关键字或保留字，它是 Verilog HDL 内部的专用词，是事先定义好的确认符，用来组织语言结构。用户不能随便使用这些关键字。需注意的是，所有关键字都是小写的。例如，ALWAYS 不是关键字，它只是标识符，与 always(关键字)

是不同的。表 2.1-1 所示为 Verilog HDL 的常用关键字。

表 2.1-1 Verilog HDL 中的常用关键字

always	end	initial	parameter	signed	triand
and	endattribute	inout	pmos	small	trior
assign	endcase	input	posedge	specify	trireg
attribute	endmodule	integer	primitive	specparam	unsigned
begin	endfunction	join	pull0	strength	vectored
buf	endprimitive	large	pull1	strong0	wait
bufif0	endspecify	macromodule	pullup	strong1	wand
bufif1	endtable	medium	pulldown	supply0	weak0
case	endtask	module	rcmos	supply1	weak1
casex	event	nand	real	table	while
casez	for	negedge	realtime	task	wire
cmos	force	nmos	reg	time	wor
deassign	forever	nor	release	tran	xnor
default	fork	not	repeat	tranif0	xor
defparam	function	notif0	rtran	tranif1	
disable	highz0	notif1	rtranif0	tri	
edge	highz1	or	rtranif1	tri0	
else	if	output	scalared	tri1	

2.1.5 数值

Verilog HDL 有四种基本的逻辑数值状态，用数字或字符表达数字电路中传送的逻辑状态和存储信息。Verilog HDL 逻辑数值中，x 和 z 都不区分大小写，也就是说，0x1z 与值 0X1Z 是等同的。Verilog HDL 中的四值电平逻辑如表 2.1-2 所示。

表 2.1-2 四值电平逻辑

状　态	含　义
0	低电平、逻辑 0 或“假”
1	高电平、逻辑 1 或“真”
x 或 X	不确定或未知的逻辑状态
z 或 Z	高阻态

在数值中，下划线符号“_”除了不能放于数值的首位外，可以随意用在整型数与实型数中，它们对数值大小没有任何改变，只是为了提高可读性。例如，16'b1011000110001100 和 16'b1011_0001_1000_1100 的数值大小是相同的，只是后一种的表达方式可读性更强。

1．整数及其表示

Verilog HDL 中的整数可以是二进制(b 或 B)、八进制(o 或 O)、十进制(d 或 D)与十六进制(h 或 H)，其基数符号与可以采用的数字字符集如表 2.1-3 所示。

表 2.1-3　数制的基数符号与数字字符集

数　制	基数符号	数字字符集
二进制	b 或 B	0、1、x、X、z、Z、?、_
八进制	o 或 O	0~7、x、X、z、Z、?、_
十进制	d 或 D	0~9、_
十六进制	h 或 H	0~9、a~f、A~F、x、X、z、Z、?、_

整数的表示形式如下：

+/−<size>'<base_format><number>

其中，“+/−”是正数和负数标示；size 指换算过后的二进制数的宽度；“ ' ”为基数格式表示的固有字符，该字符不能缺省，否则为非法表示形式；base_format 是其基数符号；number 是可以使用的数字字符集，形式上是相应进制格式下的一串数值。

使用整数时需要注意的是：

(1) 在较长的数之间可以用下划线来分开，目的是提高可读性，下划线本身没有意义，如 16'b1110_1011_0011_1010，但下划线符号不能用作首字符。

(2) 当数字没有说明位宽时，默认为 32 位。

(3) x 或 z 在二进制中代表 1 位 x 或 z，在八进制中代表 3 位 x 或 z，在十六进制中代表 4 位 x 或 z，其代表的宽度取决于所用的进制。例如：

```
8'b1011xxxx          //等价于 8'hBx
8'b1001zzzz          //等价于 8'h9z
```

(4) 若没有定义一个整数的位宽，其宽度为相应值中定义的位数。例如：

```
'o642                //9 位八进制数
'hBD                 //8 位十六进制数
```

(5) 若定义的位宽比实际数的位数大，则在左边用 0 补齐。但如果数最左边一位为 x 或 z，就相应地用 x 或 z 在左边补齐。例如：

```
10'b101              //左边补 0，得 0000000101
8'bz0x1              //左边补 z，得 zzzzz0x1
```

如果定义的位宽比实际数的位宽小，那么最左边的位被截断。例如：

```
4'b10111011          //等价于 4'b1011
6'HFFFB              //等价于 6'H3B
```

(6) “?”是高阻态 z 的另一种表示符号。在数字的表示中，字符“?”和 Z 或 z 是等价的，可互相替代。例如：4'b???? 等价于 4'bzzzz。

(7) 整数可以带正、负号，并且正、负号应写在最左边。负数表示为二进制的补码形式。例如：−4 等价于 4'b1100。

(8) 如果位宽和进制都缺省，则代表十进制数。例如：−15 代表十进制数 −15。

(9) 数字中不能有空格，但在表示进制的字母两侧可以有空格。

例 2.1-6　下面是一些合法的整数表示。

```
8'b10001101          //位宽为 8 位的二进制数 10001101
```

```
8'ha6                //位宽为 8 位的十六进制数 a6
5'o35                //5 位八进制数 35
4'd6                 //4 位十进制数 6
4'b1x_01             //4 位二进制数 1x01
5'hx                 //5 位十六进制数 x (扩展的 x)，即 xxxxx
4'hz                 //4 位十六进制数 z，即 zzzz
8  'h  2A            //在位宽和字符之间以及进制和数值之间可以有空格，但数字之间不能有空格
```

例 2.1-7　下面是错误的整数表示。

```
4'd-4                //数值不能为负，有负号应放最左边
3'  b001             //' 和基数 b 之间不允许出现空格
(4+4) 'b11           //位宽不能是表达式形式
```

2. 实数及其表示

实数有两种表示方法。

(1) 十进制表示法。采用十进制格式，小数点两边必须都有数字，否则为非法的表示形式。例如：3.0、4.54、0.2 等都是正确的，而 5.是错误的。

(2) 科学计数法。例如：564.2e2 的值为 56420.0，8.7E2 的值为 870.0(e 不分大小写)，3E−3 的值为 0.003。

Verilog HDL 还定义了实数转换为整数的方法，实数通过四舍五入转换为最相近的整数。例如：−13.74 转换为整数是 −14，33.27 转换为整数是 33。

例 2.1-8　实数表示示例。

```
2.7                  //十进制计数法
5.2e8                //科学计数法
3.5E-6               //科学计数法可用 e 或 E 表示，其结果相同
5_4582.2158_5896     //使用下划线提高可读性
6.                   //非法表示
.3e5                 //非法表示
```

3. 字符串及其表示

字符串是指用双引号括起来的字符序列，它必须包含在同一行中，不能分行书写。若字符串用作 Verilog HDL 表达式或赋值语句中的操作数，则字符串被看做 8 位的 ASCII 值序列，即一个字符对应 8 位的 ASCII 值。例如"hello world"和"An example for Verilog HDL"是标准的字符串类型。

2.2 数据类型

在 Verilog HDL 中，数据类型共有 19 种，分别是 wire、tri、tri0、tri1、wand、triand、trireg、trior、wor、reg、large、small、scalared、medium、vectored、integer、time、real、parameter 型。

按照抽象程度，Verilog HDL 的数据类型又可划分为两大类：物理数据类型(主要包括连

线型及寄存器型)和抽象数据类型(主要包括整型、时间型、实型及参数型)。

物理数据类型的抽象程度比较低，与实际硬件电路的映射关系比较明显；而抽象数据类型则是进行辅助设计和验证的数据类型。

2.2.1 物理数据类型

Verilog HDL 最主要的物理数据类型是连线型、寄存器型和存储器型，并使用四种逻辑电平和八种信号强度对实际的硬件电路建模。四值逻辑电平是对信号的抽象表示方式(见表 2.1-2)。信号强度表示数字电路中不同强度的驱动源，用来解决不同驱动强度下的赋值冲突，逻辑 0 和 1 可以用表 2.2-1 列出的强度值表示，驱动强度从 supply 到 highz 依次递减。

表 2.2-1 强 度 等 级

标 记 符	名 称	类 型	强弱程度
supply	电源级驱动	驱动	最强
strong	强驱动	驱动	↓
pull	上拉级驱动	驱动	
large	大容性	存储	
weak	弱驱动	驱动	
medium	中性驱动	存储	
small	小容性	存储	
highz	高容性	高阻	最弱

1. 连线型

连线型变量包含多种类型，表 2.2-2 给出了各种连线型变量类型以及相应的含义。

表 2.2-2 连线型数据类型及其功能说明

连线型数据类型	功 能 说 明
wire，tri	标准连线型(缺省为该类型)
wor，trior	多重驱动时，具有线或特性的连线型
wand，trand	多重驱动时，具有线与特性的连线型
trireg	具有电荷保持特性的连线型(特例)
tri1	上拉电阻
tri0	下拉电阻
supply1	电源线，用于对电源建模，为高电平 1
supply0	电源线，用于对“地”建模，为低电平 0

连线表示逻辑单元的物理连接，可以对应为电路中的物理信号连线，这种变量类型不能保持电荷(除 trireg 之外)。连线型变量必须要有驱动源，一种是连接到一个门或者模块的输出端，另一种是用 assign 连续赋值语句对它进行赋值。若没有驱动源，将保持高阻态 z。

1) wire 和 tri

在众多的连线型数据类型中，最常见的是 wire(连线)和 tri(三态线)两种，它们的语法和语义一致。不同之处在于：wire 型变量通常用来表示单个门驱动或连续赋值语句驱动的连

线型数据，tri 型变量则用来表示多驱动器驱动的连线型数据，主要用于定义三态的线网。wire/tri 的真值表如表 2.2-3 所示。

表 2.2-3　wire/tri 的真值表

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

上述真值表可以理解为，同时有两个驱动强度相同的驱动源来驱动 wire 或 tri 变量时的输出结果。

例 2.2-1　连线型变量示例。

```
wire [2:0]c;
assign c = 3'b0x1;
assign c = 3'b1z1;   // c = 3'bxx1;
```

在上面的例子中，两个驱动源共同决定 c 的值。第一个赋值语句右侧表达式的值为 0x1，第二个赋值语句右侧表达式的值为 1z1，那么 c 的有效值是 xx1。这是因为 0 和 1 同时驱动的结果为 x，x 和 z 同时驱动的结果为 x，而 1 和 1 同时驱动的结果为 1。

2)　wor 和 trior

当有多个驱动源驱动 wor 和 trior 型数据时，将产生线或结构，其真值表如表 2.2-4 所示。

表 2.2-4　wor/trior 的真值表

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

3)　wand 和 triand

当有多个驱动源驱动 wand 和 triand 型数据时，将产生线与结构，其真值表如表 2.2-5 所示。

表 2.2-5　wand/triand 的真值表

wand/triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

4)　tri0 和 tri1

tri0(tri1)的特征是：若无驱动源驱动，其值为 0(tri1 的值为 1)。在多个驱动源情况下，tri0/tri1 的真值表如表 2.2-6 所示。

表 2.2-6 tri0/tri1 的真值表

tri0/tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0/1

5) supply0 和 supply1

supply0 用于对“地”建模，即低电平 0；supply1 用于对电源建模，即高电平 1。例如：supply0 表示 Gnd，supply1 表示 Vcc。

6) trireg 线网

trireg 线网能存储数值(类似于寄存器型数据类型)，并且用于电容节点的建模。当三态寄存器(trireg)的所有驱动源都处于高阻态 z 时，三态寄存器线网将保存作用在线网上的最后一个逻辑值。三态寄存器线网的缺省初始值为 x。

trireg 网络型数据用于模拟电荷存储。电荷量强度可由下面的关键字来控制：small、medium、large。默认的电荷强度为 medium。一个 trireg 网络型数据能够模拟一个电荷存储节点，该节点的电荷量将随时间而逐渐衰减。对于一个 trireg 网络型数据，仿真时其电荷衰减时间应当制定为延迟时间。

2. 寄存器型

reg 型变量是最常见也是最重要的寄存器型数据类型，它是数据储存单元的抽象类型，其对应的硬件电路元件具有状态保持作用，能够存储数据，如触发器、锁存器等。reg 型变量常用于行为级描述中，由过程赋值语句对其进行赋值。

reg 型数据与 wire 型数据的区别在于，reg 型数据保持最后一次的赋值，而 wire 型数据需要有持续的驱动。一般情况下，reg 型数据的默认初始值为不定值 x，缺省时的位宽为 1 位。

reg 型变量举例：

```
reg a;                  //定义一个 1 位的名为 a 的 reg 型变量
reg [3:0] b;            //定义一个 4 位的名为 b 的 reg 型变量
reg[8:1]c, d, e;              //定义三个名称分别为 c、d、e 的 8 位的 reg 型变量
```

reg 型变量一般为无符号数，若将一个负数赋给 reg 型变量，则自动转换成其二进制补码形式。例如：

```
reg signed[3:0] rega;
rega = -2;                    //rega 的值为 1110(14)，是 2 的补码
```

在过程块内被赋值的每一个信号都必须定义成 reg 型，并且只能在 always 或 initial 过程块中赋值，大多数 reg 型信号常常是寄存器或触发器的输出。

2.2.2 连线型和寄存器型数据类型的声明

1. 连线型数据类型的声明

缺省的连线型数据的默认类型为 1 位(标量)wire 类型。Verilog HDL 禁止对已经声明过的网络、变量或参数再次声明。连线型数据类型声明的一般语法格式如下：

```
<net_declaration> <drive_strength><range><delay>[list_of_variables];
```

其中，drive_strength、range、delay 为可选项，而 list_of_variables 为必选项。

说明：

(1) net_ declaration：网络型数据类型，可以是 wire、tri、tri0、tri1、wand、triand、trior、wor、trireg 中的任意一种。对于 trireg 类型，其声明还有一个 charge_strength(电荷强度)的可选项。

(2) drive_strength：连线变量的驱动强度。

(3) range：指定数据为标量或矢量。若该项默认，表示数据类型为 1 位的标量，超过 1 位则为矢量形式。

(4) delay：指定仿真延迟时间。

(5) list_of_variables：变量名称，一次可定义多个名称，之间用逗号分开。

除了逻辑值外，Verilog HDL 还用强度值来解决数字电路中不同强度的驱动源之间的赋值冲突，强度等级说明见表 2.2-1。如果两个具有不同强度的信号驱动同一个线网，则竞争结果为高强度信号的值；如果两个强度相同的信号之间发生竞争，则结果为不确定值。

2. 寄存器型数据类型的声明

reg 型数据类型声明的一般语法格式如下：

```
reg<range><list_of_register_variables>;
```

其中，range 为可选项，它指定了 reg 型变量的位宽，缺省时为 1 位。

说明：

list_of_register_variables：变量名称列表，一次可以定义多个名称，之间用逗号分开。

3. 物理数据类型声明示例

前面已经了解了连线型数据和寄存器型数据这两种物理数据类型的声明格式，下面举例来理解这两种声明格式的用法。

例 2.2-2 物理数据类型声明示例。

```
reg rega;                        //定义一个 1 位的寄存器型变量
reg[7:0] regb;                   //定义一个 8 位的寄存器型变量
tri[7:0] tribus;                 //定义一个 8 位的三态总线
tri0[15:0] busa;                 //定义一个 16 位的连线型变量，处于三态时为下拉电阻
tri1[31:0] busb;                 //定义一个 32 位的连线型变量，处于三态时为上拉电阻
reg scalared[1:4]b;              //定义一个 4 位的标量型寄存器矢量
wire(pull1, strong0)c = a+b;     //定义一个 1 和 0 的驱动强度不同的 1 位连线型变量 c
trireg(large) storeline;         //定义一个具有大强度的电荷存储功能的存储线
```

2.2.3 存储器型

存储器型(memory)本质上还是寄存器型变量阵列，只是 Verilog HDL 中没有多维数组，所以就用 reg 型变量建立寄存器组(数组)来实现存储器的功能，也就是扩展的 reg 型数据地址范围。存储器型变量可以描述 RAM 型、ROM 型存储器以及 reg 文件。数组中的每一个单元通过一个数组索引进行寻址。

存储器型变量的一般声明格式如下：

```
reg <range1><name_of_register><range2>;
```

其中，range1 和 range2 都是可选项，缺省时都为 1。

说明：

(1) range1：存储器中寄存器的位宽，格式为[msb:lsb]。

(2) range2：寄存器的个数，格式为[msb:lsb]，即有 msb−lsb+1 个。

(3) name_of_register：变量名称列表，一次可以定义多个名称，之间用逗号分开。

例 2.2-3 存储器型变量声明。

```
reg[7:0] mem1[255:0];   //定义了一个有 256 个 8 位寄存器的存储器 mem1 地址范围是 0 到 255
reg[15:0]mem2[127:0], reg1, reg2;  //定义了一个具有 128 个 16 位寄存器的存储器 mem2 和
                                    //2 个 16 位的寄存器 reg1 和 reg2
```

注意 memory 型和 reg 型数据的区别。一个由 n 个 1 位寄存器构成的存储器和一个 n 位寄存器的意义是不同的。

例 2.2-4 存储器型变量与寄存器型变量的比较。

```
reg[n-1:0] a;          //表示一个 n 位的寄存器 a
reg mem1[n-1:0];       //表示一个由 n 个 1 位寄存器构成的存储器 mem1
```

一个 n 位的寄存器可以在一条赋值语句里进行赋值，而一个完整的存储器则不行。例如，对于上例可以进行“reg a = 0;”的赋值操作，而不能进行“mem1 = 0;”的赋值操作。

如果想对存储器中的存储单元进行读写操作，则必须指定该单元在存储器中的地址。

例 2.2-5 存储器型变量的赋值。

```
mem1[2] = 0;           //给 mem1 存储器中的第 3 个存储单元(寄存器)赋值为 0
```

另外，进行寻址的地址索引可以是表达式，这样就可以对存储器中的不同单元进行操作。

2.2.4 抽象数据类型

除了物理数据类型外，Verilog HDL 还提供了以下几种抽象数据类型：整型(integer)、时间型(time)、实型(real)及参数型(parameter)。它们只是纯数学的抽象描述，不能够与实际的硬件电路相映射。

1. 整型

整型数据常用于对循环控制变量的说明，在算术运算中被视为二进制补码形式的有符号数。除了寄存器型数据被当作无符号数来处理之外，整型数据与 32 位寄存器型数据在实际意义上相同。

整型数据的声明格式如下：

```
integer<list_of_register_variables>;
```

例 2.2-6 整型数据声明。

```
integer index;         //简单的 32 位有符号整数
integer i[31:0]        //定义了整型数组，它有 32 个元素
```

2. 时间型

时间型数据与整型数据类似，只是它是 64 位的无符号数。时间型数据主要用于对模拟

时间的存储与计算处理，常与系统函数$time 一起使用。

时间型数据的声明格式如下：

```
time<list_of_register_variables>;
```

例如：

```
time a, b;                    //定义了两个 64 位的时间型变量
```

3．实型

Verilog HDL 支持实型常量与变量。实型数据在机器码表示法中是浮点型数值，可用于对延迟时间的计算。

实型数据的声明格式如下：

```
real<list_of_variables>;
```

例如：

```
real stime;               //定义了一个实型数据
```

4．参数型

在 Verilog HDL 中，参数是一个非常重要的数据类型，属于常量，在仿真开始之前就被赋值，在仿真过程中保持不变。采用参数定义方法可以提高程序的可读性和可维护性。参数常用来定义延迟时间和变量的位宽。

参数类型的定义格式如下：

parameter 参数名 1 = 表达式 1，参数名 2 = 表达式 2，…，参数名 n = 表达式 n;

其中，表达式既可以是常数，也可以是表达式。参数定义完以后，程序中出现的所有的参数名都将被替换为相对应的表达式。

例 2.2-7　参数的定义。

```
parameter length = 32, weight = 16;
parameter PI = 3.14, LOAD = 4'b1101;
parameter DELAY = (BYTE+BIT)/2;
```

另外，对于同一个模块来说，参数一旦被定义就不能够通过其它语句对它重新赋值。下例是错误的，因为 a 被重复赋值。

```
module para1(…);
        parameter a = 1, b = 2;
        …
        if(…)   a = 3;
        …
endmodule
```

若要改变参量的值，可通过模块之间的参数传递来实现，这将在后面模块的引用部分详细介绍。

2.3 运 算 符

Verilog HDL 的运算符主要针对数字逻辑电路制定，覆盖范围广泛。语法规定的运算符

及其运算优先级如表 2.3-1 所示。不同的综合开发工具在执行这些优先级时可能有微小的差别，因此在书写程序时建议用括号来控制运算的优先级，以有效避免错误，同时增加程序的可读性。Verilog HDL 中的运算符和优先级如表 2.3-1 所示。

表 2.3-1　Verilog HDL 中的运算符和优先级

<table>
<tr><th>运 算 符</th><th>功　能</th><th>优 先 级 别</th></tr>
<tr><td>! 、~</td><td>反逻辑、位反相</td><td rowspan="12">高
↓
低</td></tr>
<tr><td>*、 / 、%</td><td>乘、除、取模</td></tr>
<tr><td>+ 、-</td><td>加、减</td></tr>
<tr><td><< 、 >></td><td>左移、右移</td></tr>
<tr><td>< 、<= 、> 、>=</td><td>小于、小于等于、大于、大于等于</td></tr>
<tr><td>==、!=、===、!==</td><td>等、不等、全等、非全等</td></tr>
<tr><td>&</td><td>按位与</td></tr>
<tr><td>^ 、^~</td><td>按位逻辑异或和同或</td></tr>
<tr><td>|</td><td>按位逻辑或</td></tr>
<tr><td>&&</td><td>逻辑与</td></tr>
<tr><td>||</td><td>逻辑或</td></tr>
<tr><td>?:</td><td>条件运算符，唯一的三目运算符，等同于 if-else</td></tr>
</table>

2.3.1　算术运算符

Verilog HDL 中常用的算术运算符主要有五种，分别是加法(+)、减法(-)、乘法(*)、除法(/)和取模(%)。

这五种运算符都属于双目运算符。符号“+”、“-”、“*”、“/”分别表示常用的加、减、乘、除四则运算；% 是取模运算，如“6%3”的值为 0，“7%4”的值为 3。

在算术运算符的使用中，要注意如下问题：

(1) 算术运算结果的位宽。算术表达式结果的长度由最长的操作数决定。在赋值语句下，算术运算结果的长度由等号操作符左端的目标长度决定。

例 2.3-1　算术运算示例。

```
reg[3:0]A, B, C;
reg[5:0]D;
A = B+C;
D = B+C;
```

第一个加法中，表达式“B + C”的位宽由 B、C 中最长的位宽决定，为 4 位，结果位宽由 A 决定，为 4 位；第二个加法中，右端表达式的位宽同样由 B、C 中最长的位宽决定，为 4 位，但结果的位宽由 D 决定，为 6 位。在第一个赋值中，加法操作的溢出部分被丢弃；而在第二个赋值中，任何溢出的位存储在 D[5:4]中。

(2) 有符号数和无符号数的使用。在设计中，要注意到哪些操作数应该是无符号数，哪些应该是有符号数。

无符号数值一般存储在线网、reg(寄存器)型变量及普通(没有符号标记 s)的基数格式表示的整型数中。

有符号数值一般存储在整型变量、十进制形式的整数、有符号的 reg(寄存器)型变量及有符号的线网中。

例 2.3-2　算术运算程序示例。

```
module arith_tb;
    reg[3:0]a;
    reg[2:0]b;
    initial
            begin
                a = 4'b1111;                //15
                b = 4'b011;             //3
                $display("%b", a*b);  //乘法运算，结果为 4'b1101，高位被舍去
                                      //等于 45 的低四位
                $display("%b", a/b);  //除法运算，结果为 4'b0101
                $display("%b", a+b);  //加法运算，结果为 4'b0010
                $display("%b", a-b);  //减法运算，结果为 4'b1100
                $display("%b", a%b);  //取模运算，结果为 4'b0000
            end
endmodule
```

2.3.2 关系运算符

关系运算符也是双目运算符，是对两个操作数的大小进行比较。关系运算符有大于(>)、小于(<)、大于等于(>=)和小于等于(<=)几种。

在进行关系比较时，如果成立则结果为 1'b1，否则返回的结果为 0'b0；若不确定则返回结果为不定值(x)。例如：10>15 的结果为假(0)，20 > 18 的结果为真(1)，而 4'b1101 < 4'hx 的结果为不定值(x)。

需注意的是，若操作数长度不同，则长度短的操作数应在左边用 0 补齐。例如：'b1001 >= 'b101100 等价于 'b001001 >= 'b101101，结果为假(0)。

例 2.3-3　关系运算程序示例。

```
module rela_tb;
    reg[3:0]a, b, c, d;
    initial
            begin
                a = 3;
                b = 6;
                c = 1;
```

```
            d = 4'hx;
            $display(a<b);        //结果为真(1)
            $display(a>b);        //结果为假(0)
            $display(a <= c);     //结果为假(0)
            $display(d <= a);     //结果为不定值(x)
        end
endmodule
```

2.3.3 相等关系运算符

相等关系运算符是对两个操作数进行比较，比较的结果有三种，即真(1)、假(0)和不定值(x)。Verilog HDL 中有四种相等关系运算符：等于(==)、不等于(!=)、全等(===)、非全等(! ==)。

这四种运算符都是双目运算符，要求有两个操作数。并且，这四种相等运算符的优先级别是相同的。“==”和“!=”称为逻辑等式运算符，其结果由两个操作数的值决定，由于操作数中某些位可能是不定值 x 和高阻态值 z，所以结果可能为不定值 x。

“===”和“!==”运算符则不同，它是对操作数进行按位比较，两个操作数必须完全一致，其结果才是 1，否则为 0。但是，若两个操作数对应位出现不定值 x 和高阻值 z，则可认为是相同的。“===”和“!==”运算符常用于 case 表达式的判别，所以又称为“case 等式运算符”。

表 2.3-2 列出了“==”和“===”的真值表，帮助理解两者的区别。

表 2.3-2(a)　“==”运算符的真值表

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

表 2.3-2(b)　“===”运算符的真值表

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

例 2.3-4　相等关系运算示例。

```
module equal_tb;
  reg[3:0]a, b, c, d;
    initial
      begin
            a = 4'b0xx1;
            b = 4'b0xx1;
            c = 4'b0011;
            d = 2'b11;
            $display(a==b);        //结果为不定值(x)
            $display(c==d);        //结果为真(1)
            $display(a===b);       //结果为真(1)
```

```
        $display(c===d);            //结果为假(0)
    end
endmodule
```

2.3.4 逻辑运算符

逻辑运算符有三种，分别是逻辑与(&&)、逻辑或(||)、逻辑非(!)。其中逻辑与、逻辑或是双目运算符，逻辑非为单目运算符。

逻辑运算符的操作数只能是逻辑 0 或者逻辑 1。三种逻辑运算符的真值表如表 2.3-3 所示。

表 2.3-3　逻辑运算符的真值表

a	b	!a	!b	a&&b	a||b
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

在逻辑运算符的操作过程中，如果操作数是 1 位的，那么 1 就代表逻辑真，0 就代表逻辑假；如果操作数是由多位组成的，则当操作数每一位都是 0 时才是逻辑 0 值，只要有某一位为 1，这个操作数就是逻辑 1 值。例如：寄存器变量 a、b 的初值分别为 4'b1110 和 4'b0000，则 !a = 0，!b = 1，a&&b = 0；a || b = 1。

需注意的是，若操作数中存在不定态 x，则逻辑运算的结果也是不定态，例如：a 的初值为 4'b1100，b 的初值为 4'b01x0，则 !a = 0，!b = x，a&&b = x，a || b = x。

2.3.5 按位运算符

数字逻辑电路中，信号与信号之间的运算称为位运算。Verilog HDL 提供了以下五种类型的位运算符：按位取反(~)、按位与(&)、按位或(|)、按位异或(^)、按位同或(^~)。

按位运算符对其自变量的每一位进行操作，例如，表达式 A|B 的结果是 A 和 B 的对应位相或的值。表 2.3-4～表 2.3-6 给出了按位与、按位或和按位异或运算符的真值表。

表 2.3-4　按位与运算符的真值表

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

表 2.3-5　按位或运算符的真值表

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

表 2.3-6　按位异或运算符的真值表

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

需要注意的是，两个不同长度的数据进行位运算时，会自动地将两个操作数按右端对齐，位数少的操作数会在高位用 0 补齐，然后逐位进行运算，运算结果的位宽与操作数中的位宽较大者相同。

例 2.3-5　按位运算程序示例。

```
module bit_tb;
    reg[2:0]a;
    reg[4:0]b;
    initial
        begin
            a = 5'b101;              //运算的时候 a 自动变为 5'b00101
            b = 5'b11101;
            $display("%b", ~a);      //结果为 5'b11010
            $display("%b", ~b);      //结果为 5'b00010
            $display("%b", a&b);     //结果为 5'b00101
            $display("%b", a|b);     //结果为 5'b11101
            $display("%b", a^b);     //结果为 5'b11000
        end
endmodule
```

2.3.6　归约运算符

归约运算符按位进行逻辑运算，属于单目运算符。由于这一类运算符运算的结果是产生 1 位逻辑值，因而被形象地称为缩位运算符。

Verilog HDL 中，缩位运算符包括 &(与)、| (或)、^ (异或)以及相应的非操作 ~&、~|、~^、^~。归约运算符的操作数只有一个。

归约运算符的运算过程是：设 a 是一个 4 位的寄存器型变量，它的 4 位分别是 a[0]、a[1]、a[2] 和 a[3]。当对 a 进行缩位运算时，先对 a[0] 和 a[1] 进行缩位运算，产生 1 位的结果，再将这个结果与 a[2] 进行缩位运算，再接着是 a[3]，最后产生 1 位的操作结果。

例 2.3-6　归约运算程序示例。

```
module cut_tb;
    reg[5:0]a;
    initial
        begin
```

```
            a = 6'b101011;
            $display("%b", &a);          //结果为 1'b0
            $display("%b", |a);          //结果为 1'b1
            $display("%b", ^a);          //结果为 1'b0
        end
endmodule
```

2.3.7 移位运算符

移位运算符有两种：左移位运算符(<<)、右移位运算符(>>)。运算过程是将左边(右边)的操作数向左(右)移，所移动的位数由右边的操作数来决定，然后用 0 来填补移出的空位。

例 2.3-7 移位运算程序示例。

```
module shift_tb;
    reg[5:0]a, b, c, d;
    reg[7:0]e;
    initial
      begin
            a = 6'b101101;
            b = a<<2;
            c = a>>3;
            d = a<<7;
            e = a<<2;
            $display("%b", b);     //结果为 6'b110100
            $display("%b", c);     //结果为 6'b000101
            $display("%b", d);     //结果为 6'b000000
            $display("%b", e);     //结果为 8'b10110100
      end
endmodule
```

从上例可以看出，a 在移位后，用 0 来填补了空出的位。进行移位运算时应当注意移位前后变量的位数。

2.3.8 条件运算符

条件运算符(?:)是 Verilog HDL 里唯一的三目运算符，它根据条件表达式的值来选择应执行的表达式，其表达形式如下：

<条件表达式>?<表达式 1>:<表达式 2>

其中，条件表达式的运算结果有真(1)、假(0)和不定态(x)三种。当条件表达式的结果为真时，执行表达式 1，当条件表达式的结果为假时，执行表达式 2。

如果条件表达式的运算结果为不定态 x，则模拟器将按位对表达式 1 的值与表达式 2 的值进行比较，位与位的比较按表 2.3-7 的规则产生每个结果位，从而构成条件表达式的结果值。

表 2.3-7　条件表达式为不定态时的结果产生规则

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

例 2.3-8　条件运算程序示例。

```
module mux2(in1, in2, sel, out);
    input[3:0]in1, in2;
    input sel;
    output[3:0]out;
    wire[3:0]out;
    assign out = (!sel)?in1:in2;    //sel 为 0 时 out 等于 in1，反之为 in2
endmodule
```

例 2.3-8 描述了一个 2 选 1 的数据选择器，图 2.3-1 是其电路结构。

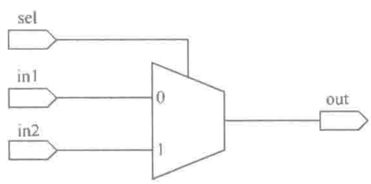


图 2.3-1　2 选 1 数据选择器电路结构

若该数据选择器的 sel 端为不定态 x，则 out 由 in1 和 in2 按位运算的结果得出。若 in1 = 4'b0011，in2 = 4'b0101，则按照上述真值表得出 out = 4'b0xx1。

2.3.9　连接和复制运算符

Verilog HDL 中还有两个特殊的运算符：连接运算符({})和复制运算符({{}})。

连接运算符是把位于大括号({})中的两个或两个以上信号或数值用逗号(,)分隔的小表达式按位连接在一起，最后用大括号括起来表示一个整体信号，形成一个大的表达式。其格式如下：

{信号 1 的某几位，信号 2 的某几位，…，信号 n 的某几位}

重复运算符({{}})将一个表达式放入双重花括号中，复制因子放在第一层括号中。它为复制一个常量或变量提供了一种简便的方法。

例 2.3-9　连接和复制运算程序示例。

```
module con_rep_tb;
  reg [2:0]a;
  reg [3:0]b;
  reg [7:0]c;
  reg [4:0]d;
```

```
    reg [5:0]e;
    initial
        begin
            a = 3'b101;
            b = 4'b1110;
            c = {a, b};                     //连接操作
            d = {a[2:1], b[2:0]};           //连接操作
            e = {2{a}};                     //复制操作
            $display("%b", c);              //结果为 8'b01011110
            $display("%b", d);              //结果为 5'b10110
            $display("%b", e);              //结果为 6'b101101
        end
endmodule
```

2.4 模　　块

2.4.1 模块的基本概念

模块(module)是 Verilog HDL 的基本单元，它代表一个基本的功能块，用于描述某个设计的功能或结构，以及与其它模块通信的外部端口。一个电路的设计不仅仅局限于一个模块的设计，而是由多个模块组合而成的，因此一个模块的设计只是一个系统设计中某个层次的设计。

图 2.4-1 是一个基本的模块结构组成。



图 2.4-1　基本的模块结构组成

从图 2.4-1 可以看出，一个模块主要包括模块的开始与结束、模块端口定义、模块数据类型说明和模块逻辑功能描述几个基本部分。

(1) 模块的开始与结束：模块在语言形式上是以关键词 module 开始、以关键词 endmodule 结束的一段程序，其中模块开始语句必须以分号结束。模块的开始部分包括模块名(name)和端口列表(port_list)，模块名是模块唯一性的标识符，而端口列表是由模块各个输入、输出和双向端口变量组成的一张列表，这些端口用来与其它模块进行连接(不妨理解为集成电路的引脚)。

(2) 模块端口定义：定义端口列表里的变量哪些是输入(input)、输出(output)和双向端口(inout)以及位宽。

(3) 模块数据类型说明：包括 wire、reg、memory 和 parameter 等数据类型，以说明模块中所用到的内部信号、调用模块等的声明语句和功能定义语句。一般来说，module 的 input 缺省定义为 wire 类型；output 信号可以是 wire 类型，也可以是 reg 类型(条件是在 always 或 initial 语句块中被赋值)；inout 一般为 tri(三态线)类型，表示有多个驱动源。

(4) 模块逻辑功能描述：产生各种逻辑(主要是组合逻辑和时序逻辑)，主要包括 initial 语句、always 语句、其它子模块实例化语句、门实例化语句、用户自定义原语(UDP)实例化语句、连续赋值语句(assign)、函数(function)和任务(task)。

因此由上述模块的结构组成可以看出，模块在概念上可等同于一个器件，比如通用器件(与门、三态门等)或通用宏单元(计数器、ALU、CPU)等。一个模块可在另一个模块中被调用，一个模块代表了一个特定功能块。

一个电路设计可由多个模块组合而成，因此一个模块的设计只是一个系统设计中某个层次的设计。模块设计可采用多种建模方式，一般包括行为描述方式、结构描述方式以及混合描述方式(混合使用结构描述和行为描述)。

通过下例的 Verilog HDL 设计的简单模块，可结合模块结构体会 Verilog HDL 代码和电路图的含义。

例 2.4-1 上升沿 D 触发器的设计。

```
module dff(din, clk, q);
        input din, clk;
        output q;
        reg q;
        always@(posedge clk)
              q <= din;
endmodule
```

其中，“module dff(din, clk, q);”和“endmodule”标志着模块的开始和结束，“input din, clk;”和“output q;”为端口定义部分，“reg q;”为数据类型说明部分，“always@(posedge clk)”和“q<=din;”为逻辑功能描述部分。

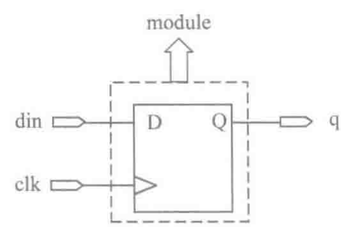


图 2.4-2 D 触发器电路

该例描述了一个简单的 D 触发器，图 2.4-2 是其电路图，虚线框里的部分可以理解为模块的概念，它是一个具有 D 触发器功能的模块，两边的 din、clk 和 q 是模块的输入和输出端口。

2.4.2 端口

1. 端口的定义

端口是模块与外界或其它模块沟通的信号线。模块的端口可以是输入端口(input)、输出端口(output)或双向端口(inout)。

缺省状态下，端口类型都默认为 wire 类型。需要注意的是，在某一端口的类型声明中，类型声明的长度必须和端口声明的长度一致。

一个模块往往具有多个端口，它们是本模块和其它模块进行联系的标志。在模块定义

格式中，“模块端口列表”列出了模块具有的外部可见端口，该“模块端口列表”内的每一个端口项都代表着一个模块端口。

Verilog HDL 中有如下三种端口声明方式。

输入端口：

input[信号位宽-1:0] 端口名 1；

输出端口：

output[信号位宽–1:0]端口名 1；

输入/输出端口：

inout[信号位宽-1:0]端口名 1；

端口声明语句的作用对象必须是与模块端口相连的模块内部变量，而不能是端口名(端口名和内部变量标识符同名的情况除外)。模块内部对端口进行的输入/输出操作是通过端口表达式中出现的内部变量进行的(这些变量与对应的端口相连)，所以在模块内部必须对这些内部变量的输入/输出特性进行说明。

2. 模块引用时端口的对应方式

当模块被引用时，在引用的模块中，有些信号要输入到被引用的模块中，有的信号需要从被引用的模块中取出。在引用模块时其端口可以用如下两种方法连接。

(1) 在引用时，严格按照模块定义的端口顺序来连接，不用标明源模块定义时规定的端口名。其格式如下：

模块名(连接端口 1 信号名，连接端口 2 信号名…)；

(2) 在引用时用“.”标明源模块定义时规定的端口名。其格式如下：

模块名(.端口 1 名(连接信号 1 名)，.端口 2 名(连接信号 2 名)…)；

这样表示的好处在于可以用端口名与被引用模块的端口对应，不必严格按端口顺序对应，提高了程序的可读性和可移植性。

模块引用属于 Verilog HDL 的结构建模部分，其详细内容将在下一章介绍。

本 章 小 结

本章主要介绍了 Verilog HDL 的基本语法。通过语言要素和基本使用方法的学习，可为后续基本逻辑电路和复杂数字电路的设计提供必要的基础。Verilog HDL 的语法、数据类型和基本设计框架具有一定的特殊性，它们与实际数字硬件电路中的组合逻辑、时序逻辑、存储器和开关等相对应。熟练掌握本章内容是学习后续章节的必要基础。

思考题和习题

1. Verilog HDL 中，基本的语言要素有哪些？
2. Verilog HDL 中，空白符总是可以忽略吗？
3. Verilog HDL 中，插入注释的方法有哪两种？
4. 下列标识符哪些是合法的，哪些是非法的？

Always，2_1mux，\din，\wait，_qout，$data，data$，c#out，\@out

5. 下列数字的表示方法是否正确？若正确，表示多少？

5'd20，'B10，4'b10x1，'dc10，5'b101，6'HAAFB

6. 已知“a = 1'b1; b = 3'b011;”那么{a, b}是多少？

7. Verilog HDL 中，数据类型通常可以分为几类？每一类又包含哪些具体的数据类型？

8. 从电路的角度分析 wire 和 reg 型数据的区别。

9. Verilog HDL 中，若连线型变量的驱动强度说明被省略，则默认的驱动强度是什么？

10. 能否对 memory 型数据进行位选择和域选择？

11. Verilog HDL 中，定义参数 parameter 有什么作用？

12. 运算符“~”和“!”、“&&”和“&”有什么区别？

13. 相等运算符(==)和全等运算符(===)有什么区别？各在什么场合使用？

14. 任意抽象的符合语法的 Verilog HDL 模块是否都可以通过综合工具转变为电路结构？

15. 一般来说，模块由哪些部分构成？每一部分又由哪些语句构成？

16. 端口有哪几种？模块的端口是如何描述的？

第 3 章　Verilog HDL 程序设计语句和描述方式

3.1　数据流建模

在数字电路中，输入信号经过组合逻辑电路传到输出时类似于数据流动，而不会在其中存储。可以通过连续赋值语句对这种特性进行建模，这种建模方式通常被称为数据流建模。

Verilog HDL 中的数据流建模方式是比较简单的行为建模，它只有一种描述方式，即通过连续赋值语句进行逻辑描述。其最基本的语句是由关键词 assign 引导的。

对于连续赋值语句，只要输入端操作数的值发生变化，该语句就重新计算并刷新赋值结果。通常可以使用连续赋值语句来描述组合逻辑电路，而不需要用门电路和互连线。连续赋值的目标类型主要是标量线网和向量线网两种，标量线网如“wire a, b;”，向量线网如“wire [3:0]a, b;”。

连续赋值语句只能用来对连线型变量进行驱动，而不能对寄存器型变量进行赋值，它可以采取显式连续赋值语句和隐式连续赋值语句两种赋值方式。

1. 显式连续赋值语句

显式连续赋值语句的语法格式如下：

```
<net_declaration><range><name>;
assign #<delay><name> = Assignment expression;
```

这种格式的连续赋值语句包含两条语句：第一条语句是对连线型变量进行类型说明的语句；第二条语句是对这个连线型变量进行连续赋值的赋值语句。赋值语句是由关键词 assign 引导的，它能够用来驱动连线型变量，而且只能对连线型变量进行赋值，主要用于对 wire 型变量的赋值。

2. 隐式连续赋值语句

隐式连续赋值语句的语法格式如下：

```
<net_declaration><drive_strength><range>#<delay><name> = assignment expression;
```

这种格式的连续赋值语句把连线型变量的说明语句以及对该连线型变量进行连续赋值的语句结合到同一条语句内。利用它可以在对连线型变量进行类型说明的同时实现连续赋值。

上述两种格式中：

- “net_declaration(连线型变量类型)”可以是除了 trireg 类型外的任何一种连线型数据类型。

● "range(变量位宽)"指明了变量数据类型的宽度，格式为[msb:lab]，缺省时为 1 位。

● "drive_strength(赋值驱动强度)"是可选的，它只能在"隐式连续赋值语句"格式中指定。它用来对连线型变量受到的驱动强度进行指定。它是由"对 1 驱动强度"和"对 0 驱动强度"两项组成的。驱动强度的概念在上一章的数据类型中已经说明，比如语句"wire(weak0，strong1)out = in1&in2；"内的"(weak0，strong1)"就表示该语句指定的连续赋值对连线型变量"out"的驱动强度是：赋"0"值时的驱动强度为"弱(weak)"，而赋"1"值时的驱动强度为"强(strong)"。如果在格式中缺省了"赋值驱动强度"这一项，则驱动强度默认为(strong1，strong0)。

● "delay(延时量)"项也是可选的，它指定了赋值表达式内信号发生变化时刻到连线型变量取值被更新时刻之间的延迟时间量。其语法格式如下：

```
#(delay1，delay2，delay3)
```

其中，delay1、delay2、delay3 都是一个数值，"delay1"指明了连线型变量转移到"1"状态时的延时值(称为上升延时)；"delay2"指明了连线型变量转移到"0"状态时的延时值(称为下降延时)；"delay3"指明了连线型变量转移到"高阻 Z"状态时的延时值(称为关闭延时)。

例 3.1-1　显式连续赋值语句示例。

```
module example1_assignment(a, b, m, n, c, y);
    input[3:0] a, b, m, n;
    output[3:0] c, y;
    wire[3:0] a, b, m, n, c, y;
    assign y = m|n;
    assign #(3, 2, 4) c = a&b;
endmodule
```

该例中包含了两个显式赋值语句，分别用来实现组合逻辑中的"或"和"与"逻辑，其赋值目标是连线型变量 c 和 y，它们的位宽都为 4 位。连续赋值语句指定用表达式"m|n"和"a&b"的取值分别对连线型变量 y 和 c 进行连续驱动。

其中，"assign y = m | n;"没有指定延时量；而"assign #(3, 2, 4) c = a&b;"指定的延时量为"(3, 2, 4)"，它指明了从信号 a 或 b 发生变化时刻到变量 c 被更新时刻之间的延迟时间量，即上升延时为 3 个时间单位，下降延时为 2 个时间单位，关闭延时为 4 个时间单位。

由于是显式赋值语句，因此并未出现"驱动强度"这一项，所以连线型变量 y 和 c 受到的驱动强度默认都是"(strong1，strong0)"。

例 3.1-2　隐式连续赋值语句示例。

```
module example2_assignment(a, b, m, n, c, y, w);
    input[3:0] a, b, m, n;
    output[3:0] c, y, w;
    wire[3:0] a, b, m, n;
    wire[3:0] y = m|n;
    wire[3:0] #(3, 2, 4) c = a&b;
    wire(strong0, weak1)[3:0] #(2, 1, 3) w = (a^b)&(m^n);
endmodule
```

由该例可以看出，在对 y 和 c 这两个变量进行隐式赋值后，其实现的组合逻辑功能与例 3.1-1 当中的显式赋值语句所实现的功能相同。另外，在对变量 w 进行隐式赋值时多了一个驱动强度的定义。对于变量 w：赋“0”值时的驱动强度较强，为 strong；赋“1”值时的驱动强度较弱，为 weak。比如，当 0 和 1 共同驱动变量 w 时，由于 0 定义的驱动强度较强，所以 w 为 0。

3. 连续赋值语句使用中的注意事项

(1) 赋值目标只能是线网类型(wire)。

(2) 在连续赋值中，只要赋值语句右边表达式任何一个变量有变化，表达式立即被计算，计算的结果立即赋给左边信号(若没有定义延时量)。

(3) 连续赋值语句不能出现在过程块中。

(4) 多个连续赋值语句之间是并行关系，因此与位置顺序无关。

(5) 连续赋值语句中的延时具有硬件电路中惯性延时的特性，任何小于其延时量的信号变化脉冲都将被滤除掉，不会出现在输出端口。

3.2 行为级建模

Verilog HDL 支持设计者从电路外部行为的角度对其进行描述，因此行为级建模是从一个层次很高的抽象角度来表示电路的。其目标不是对电路的具体硬件结构进行说明，它是为了综合以及仿真的目的而进行的。在这个层次上设计数字电路更类似于使用一些高级语言(如 C 语言)进行编程，而且 Verilog HDL 行为级建模的语法结构与 C 语言也非常相似。Verilog HDL 提供了许多行为级建模语法结构，为设计者的使用提供了很大的灵活性。

行为描述常常用于复杂数字逻辑系统的顶层设计中，也就是通过行为级建模把一个复杂的系统分解成可操作的若干个模块，每个模块之间的逻辑关系通过行为模块的仿真加以验证。这样就能把一个大的系统合理地分解为若干个较小的子系统，然后再将每个子系统用可综合风格的 Verilog HDL 模块(门级结构或 RTL 级、算法级、系统级的模块)加以描述。同时行为级建模也可以用来生成仿真测试信号，对已设计的模块进行检测。

本节将详细介绍行为级建模结构以及各种高级语句的语法格式和用法。

图 3.2-1 和表 3.2-1 分别给出了 Verilog HDL 行为描述中模块的构成框架和行为描述语句及其可综合性。

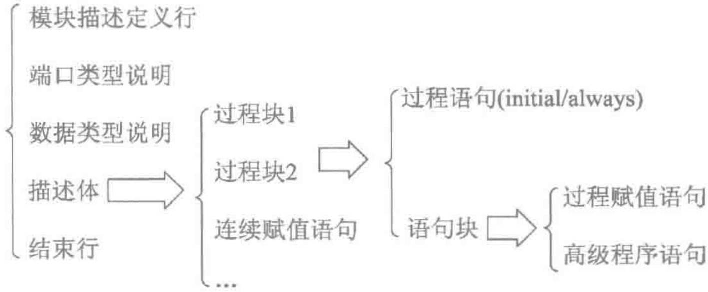


图 3.2-1 Verilog HDL 行为描述中模块的构成框架

表 3.2-1　Verilog HDL 行为描述语句及其可综合性

类　别	语　句	可综合性
过程语句	initial	
	always	√
语句块	串行语句块 begin-end	√
	并行语句块 fork-join	
赋值语句	过程连续赋值语句 assign	
	过程赋值语句 =、<=	√
条件语句	if-else	√
	case，casez，casex	√
循环语句	forever	
	repeat	
	while	
	for	√
编译向导语句	`define	√
	`include	√
	`ifdef，`else，`endif	√

3.2.1　过程语句

Verilog HDL 中的过程块是由过程语句所组成的。过程语句有两种，分别是 initial 过程语句和 always 过程语句。

1．initial 过程语句

initial 过程语句的语法格式是：

```
initial
    begin
        语句 1;
        语句 2;
        ...
        语句 n;
    end
```

initial 过程块在进行仿真时从模拟 0 时刻开始执行，它在仿真过程中只执行一次，在执行完一次后该 initial 过程块就被挂起，不再执行。如果一个模块中存在多个 initial 过程块，则每个 initial 过程块都是同时从 0 时刻开始并行执行的。initial 过程块内的多条行为语句可以是顺序执行的，也可以是并行执行的。

initial 过程语句通常用于仿真模块中对激励向量的描述，或用于给寄存器变量赋初值。

例 3.2-1　用 initial 过程语句对变量 A、B、C 进行赋值。

```
module initial_tb1;
  reg A, B, C;
```

```
    initial
        begin
                    A = 0; B = 1; C = 0;
            #100    A = 1; B = 0;
            #100    A = 0; C = 1;
            #100    B = 1;
            #100    B = 0; C = 0;
        end
endmodule
```

在 ModelSim 仿真环境下的仿真结果如图 3.2-2 所示。

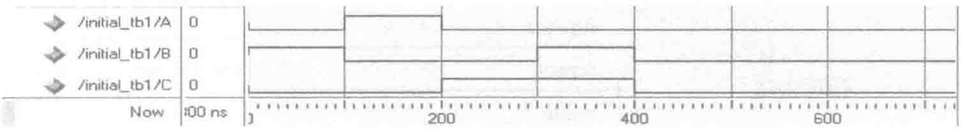


图 3.2-2　例 3.2-1 中 initial 语句赋值波形

2. always 过程语句

从语法描述角度而言，相对于 initial 过程语句，always 过程语句的触发状态是一直存在的，只要满足 always 后面的敏感事件列表，就执行语句块。其语法格式如下：

```
always@(<敏感事件列表>)
        语句块;
```

其中，敏感事件列表就是触发条件，只有当触发条件满足时，其后的语句块才能被执行。也就是说，当该列表中变量的值改变时，就会引发块内语句的执行。因此，敏感事件列表中应列出影响块内取值的所有信号。若有两个或两个以上信号，则它们之间可以用“or”连接，也可以用逗号“，”连接。敏感信号可以分为两种类型：一种为边沿敏感型，一种为电平敏感型。对于时序电路，事件通常是由时钟边沿触发的。为表达边沿这个概念，Verilog HDL 提供了 posedge 和 negedge 两个关键字分别描述信号的上升沿和下降沿。例如：

```
@(a)                            //当信号 a 的值发生改变时
@(a or b)                       //当信号 a 或信号 b 的值发生改变时
@(posedge clock)                //当 clock 的上升沿到来时
@(negedge clock)                //当 clock 的下降沿到来时
@(posedge clk or negedge reset) //当 clk 的上升沿到来或 reset 信号的下降沿(低电平有效)到来时
```

3. 过程语句使用中的注意事项

过程语句具有很强的功能，Verilog HDL 的大多数高级程序语句都是在过程中使用的。它既可以描述时序逻辑电路，也可以描述组合逻辑电路。采用过程语句进行程序设计时，Verilog HDL 有一定的设计要求和规范。

在信号的形式定义方面，无论是对时序逻辑电路还是对组合逻辑电路进行描述，Verilog HDL 要求在过程语句(initial 和 always)中，被赋值信号必须定义为“reg”类型。

敏感事件列表是 Verilog HDL 中的一个关键性设计，如何选取敏感事件作为过程的触发条件，在 Verilog HDL 程序中有一定的设计要求：

(1) 采用过程语句对组合电路进行描述时，需要把全部的输入信号列入敏感事件列表。

(2) 采用过程语句对时序电路进行描述时，需要把时间信号和部分输入信号列入敏感事件列表。

应当注意的是，不同的敏感事件列表会产生不同的电路形式。

例 3.2-2 用 initial 语句产生测试信号。

```
module initial_tb2;
        reg S1;                    //被赋值信号定义为“reg”类型
        initial
            begin
                    S1 = 0;
                #100 S1 = 1;
                #200 S1 = 0;
                #50  S1 = 1;
                #100 $finish;
            end
endmodule
```

例 3.2-3 用 always 语句描述 4 选 1 数据选择器。

4 选 1 数据选择器是一种典型的组合逻辑电路，其 Verilog HDL 程序代码如下：

```
module mux4_1(out, in0, in1, in2, in3, sel);
        output out;
        input in0, in1, in2, in3;
        input[1:0] sel;
        reg out;                                      //被赋值信号定义为“reg”类型
        always @(in0 or in1 or in2 or in3 or sel)     //敏感事件列表
            case(sel)
                2'b00:      out = in0;
                2'b01:      out = in1;
                2'b10:      out = in2;
                2'b11:      out = in3;
                default:    out = 2'bx;
            endcase
endmodule
```

上例是一个 always 块引导的电平触发事件。只要任意时刻有一个敏感事件列表里的信号发生变化，都会执行后面的 case 语句。

例 3.2-4 用 always 语句描述同步置数、同步清零计数器。

```
module counter1(out, data, load, rst, clk);
    output[7:0] out;
    input[7:0] data;
    input load, clk, rst;
```

```
    reg[7:0] out;
    always @(posedge clk)                  // clk 上升沿触发
      begin
          if(!rst) out = 8'h00;            //同步清零，低电平有效
          else if(load) out = data;        //同步置数
          else out = out + 1;
      end
  endmodule
```

例中，posedge clk 表示以时钟信号 clk 的上升沿作为触发条件。而敏感事件列表中没有列出输入信号 load、rst，这是因为它们是同步置数、同步清零，这些信号要起作用，必须等待时钟的上升沿到来。

例 3.2-5　用 always 过程语句描述异步清零计数器。

```
module counter2(rst, clk, out);
  output[7:0] out;
  input clk, rst;
  reg[7:0] out;
  always @(posedge clk or negedge rst)          // clk 上升沿和 rst 低电平清零有效
      begin
        if(!rst)                                //异步清零
          out = 0;
        else    out = out+1;
      end
endmodule
```

例中，敏感事件列表中有 posedge clk 和 negedge rst 两个敏感事件。当 negedge rst 出现时过程语句也会执行，这是对异步电路的一种直观描述，Verilog HDL 综合工具将会把这个代码综合成异步清零的电路，所选择的器件和连接关系与例 3.2-4 的结果完全不同。

3.2.2　语句块

在 Verilog HDL 过程语句的使用中，当语句数超过一条时，需要采用语句块。语句块就是由块标识符 begin-end 或 fork-join 界定的一组行为描述语句。语句块就相当于给块中的一组行为描述语句进行打包，使之在形式上类似于一条语句。语句块的具体功能是通过语句块中所包含的描述语句的执行而得以实现的。当语句块中只包含一条语句时，可以直接写这条语句，此时块标识符可以缺省。

语句块包括串行语句块(begin-end)和并行语句块(fork-join)两种。

1. 串行语句块

串行语句块采用的是关键字“begin”和“end”，其中的语句按串行方式顺序执行，可以用于可综合电路程序和仿真测试程序。其语法格式是：

```
begin:块名
    块内声明语句;
```

```
        语句 1;
        语句 2;
        ...
        语句 n;
    end
```

其中，块名即该块的名字，当块内有变量时必须有块名，否则在编译时将出现语法错误。块内声明语句是可选的，可以是参数说明语句、integer 型变量声明语句、reg 型变量声明语句、time 型变量声明语句和事件(event)说明语句。

串行语句块的特点：

(1) 串行语句块中的每条语句依据块中的排列次序逐条执行。块中每条语句给出的延迟时间都是相对于前一条语句执行结束的相对时间。

(2) 串行语句块的起始执行时间就是串行语句块中第一条语句开始执行的时间；串行语句块的结束时间就是块中最后一条语句执行结束的时间。

2. 并行语句块

并行语句块采用的是关键字“fork”和“join”，其中的语句按并行方式执行，只能用于仿真测试程序，不能用于可综合电路程序。其语法格式是：

```
fork:块名
        块内声明语句;
        语句 1;
        语句 2;
        ...
        语句 n;
join
```

并行语句块的特点：

(1) 块内语句是同时执行的，即程序流程控制一进入到该并行语句块，块内语句则同时开始执行。

(2) 块内每条语句的延迟时间是相对于程序流程控制进入到块内的仿真时间。

3. 语句块的使用

例 3.2-6　分别采用串行语句块和并行语句块产生图 3.2-3 中的信号波形。

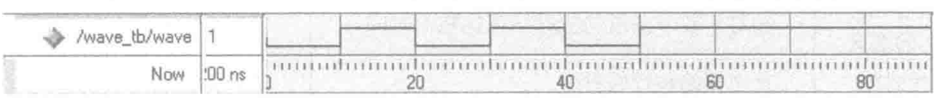


图 3.2-3　信号波形图

(1) 采用串行语句块的 Verilog HDL 程序代码如下：

```
module wave_tb1;
    reg wave;
    parameter T = 10;
      initial
          begin
```

```
                wave = 0;
            #T  wave = 1;
            #T  wave = 0;
            #T  wave = 1;
            #T  wave = 0;
            #T  wave = 1;
        end
endmodule
```

(2) 采用并行语句块的 Verilog HDL 程序代码如下：

```
module wave_tb2;
  reg wave;
  parameter T = 10;
  initial
      fork
                      wave = 0;
          #T          wave = 1;
          #(2*T)      wave = 0;
          #(3*T)      wave = 1;
          #(4*T)      wave = 0;
          #(5*T)      wave = 1;
      join
endmodule
```

从该例可以看到，采用串行语句块和并行语句块都可以产生相同的测试信号，具体采用哪种语句进行设计主要取决于设计者的习惯。需要说明的是，在对于电路的描述性设计中，部分综合工具不支持并行语句块，因此主要采用串行语句块进行设计。

表 3.2-2 对比了串行语句块和并行语句块，用以理解二者的区别和联系。

表 3.2-2　串行语句块和并行语句块对比

语句块	串行语句块(begin-end)	并行语句块(fork-join)
执行顺序	按照语句顺序执行	所有语句均在同一时刻执行
语句前面延迟时间的意义	相对于前一条语句执行结束的相对时间	相对于并行语句块启动的时间
起始时间	首句开始执行的时间	转入并行语句块的时间
结束时间	最后一条语句执行结束的时间	执行时间最长的那条语句执行结束的时间
行为描述的意义	电路中的数据在时钟及控制信号的作用下，沿数据通道中各级寄存器之间传送的过程	电路上电后，各电路模块同时开始工作的过程

3.2.3　过程赋值语句

过程块中的赋值语句称为过程赋值语句。过程性赋值是在 initial 语句或 always 语句内

的赋值，它只能对寄存器数据类型的变量赋值。对于多位宽的寄存器变量(矢量)，还可以只对其中的某一位或某几位进行赋值。对于存储器类型的，则只能通过选定的地址单元，对某个字进行赋值。还可以将前述各类变量用连接符拼接起来，构成一个整体作为过程赋值语句的左端。

过程赋值语句有阻塞赋值语句和非阻塞赋值语句两种。

1. 阻塞赋值语句

阻塞赋值语句的操作符号为“=”，其语法格式是：

```
变量 = 表达式;
```

例如：

```
b = a;
```

当一个语句块中有多条阻塞赋值语句时，如果前面的赋值语句没有完成，则后面的语句就不能被执行，仿佛被阻塞了一样，因此称为阻塞赋值方式。

阻塞赋值语句的特点：

(1) 在串行语句块中，各条阻塞赋值语句将按照排列顺序依次执行；在并行语句块中的各条阻塞赋值语句则同时执行，没有先后之分。

(2) 执行阻塞赋值语句的顺序是，先计算等号右端表达式的值，然后立刻将计算的值赋给左边的变量，与仿真时间无关。

2. 非阻塞赋值语句

非阻塞赋值语句的操作符号为“<=”，其语法格式是：

```
变量 <= 表达式;
```

例如：

```
b <= a;
```

如果在一个语句块中有多条非阻塞赋值语句，则后面语句的执行不会受到前面语句的限制，因此称为非阻塞赋值方式。

非阻塞赋值语句的特点：

(1) 在串行语句块中，各条非阻塞赋值语句的执行没有先后之分，排在前面的语句不会影响到后面语句的执行，各条语句并行执行。

(2) 执行非阻塞赋值语句的顺序是，先计算右端表达式的值，然后等待延迟时间的结束，再将计算的值赋给左边的变量。

阻塞赋值语句和非阻塞赋值语句可以用于数字逻辑电路设计和测试仿真程序中。在数字逻辑电路设计中，阻塞赋值语句和非阻塞赋值语句对于电路的描述差别很大，使用不同的赋值语句，产生的电路差异可能很大。

例 3.2-7　试分析下面两段 Verilog HDL 程序所描述的电路结构。

程序(1)：

```
module block1(din, clk, out1, out2);
    input din, clk;
    output out1, out2;
    reg out1, out2;
    always@(posedge clk)
```

```
        begin
            out1 = din;
            out2 = out1;
        end
endmodule
```

程序(2)：

```
module non_block1(din, clk, out1, out2);
    input din, clk;
    output out1, out2;
    reg out1, out2;
    always@(posedge clk)
        begin
            out1 <= din;
            out2 <= out1;
        end
endmodule
```

在这两个程序中，基本描述相同，不同的是程序(1)采用了阻塞赋值语句，而程序(2)采用了非阻塞赋值语句。

在执行阻塞赋值语句的过程中，din 的值先传给 out1，然后 out1 的值再传给 out2，等价于：

```
out1 = din;
out2 = din;
```

因此，程序(1)描述了一个寄存器，其电路结构如图 3.2-4 所示。

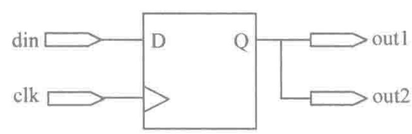


图 3.2-4　程序(1)的电路结构

在执行非阻塞赋值语句的过程中，din 的值传给 out1，同时 out1 的值传给 out2。因此，程序(2)描述了 2 个寄存器，其电路结构如图 3.2-5 所示。

如果采用阻塞赋值语句描述图 3.2-5 所示电路，则其 Verilog HDL 程序代码如下：

```
module block2(din, clk, out1, out2);
    input din, clk;
    output out1, out2;
    reg out1, out2;
    always@(posedge clk)
        begin
            out2 = out1;
            out1 = din;
        end
endmodule
```

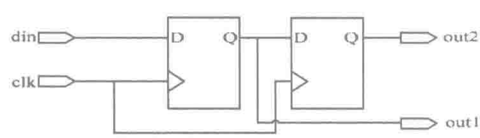


图 3.2-5　程序(2)的电路结构

可以看到，这两种赋值语句在 Verilog HDL 程序设计中的方式是不一样的，因此在使用时要仔细考虑电路的结构，选用合适的赋值方式。下面再举一个较为复杂的例子，帮助进

一步理解阻塞赋值语句和非阻塞赋值语句的使用技巧。

例 3.2-8　试分析下面两段 Verilog HDL 程序所描述的电路结构。

程序(1)：

```
module block3(a, b, c, clk, sel, out);
    input a, b, c, clk, sel;
    output out;
    reg out, temp;
    always@(posedge clk)
        begin
            temp = a&b;
            if(sel) out = temp|c;
            else out = c;
        end
endmodule
```

程序(2)：

```
module non_block2(a, b, c, clk, sel, out);
    input a, b, c, clk, sel;
    output out;
    reg out, temp;
    always@(posedge clk)
        begin
            temp <= a&b;
            if(sel) out <= temp|c;
            else out <= c;
        end
endmodule
```

程序(1)和程序(2)分别采用了阻塞赋值语句和非阻塞赋值语句，所对应的电路分别如图 3.2-6 和图 3.2-7 所示。程序(2)采用了非阻塞赋值语句，实际上产生的是两级流水线的设计。虽然采用这两种语句的逻辑功能相同，但是电路的时序和形式差异很大，这一点也是初学者应该注意的地方。

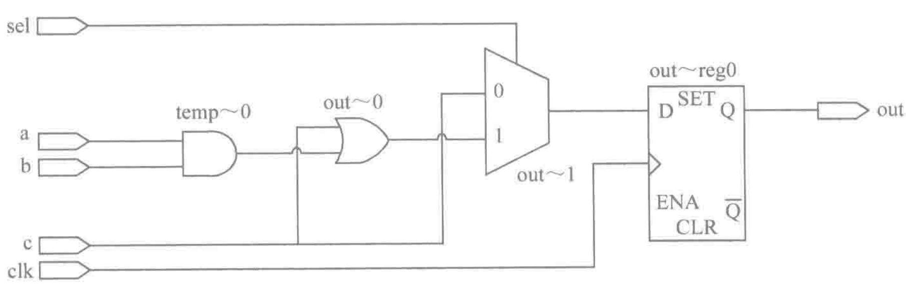


图 3.2-6　程序(1)的电路结构

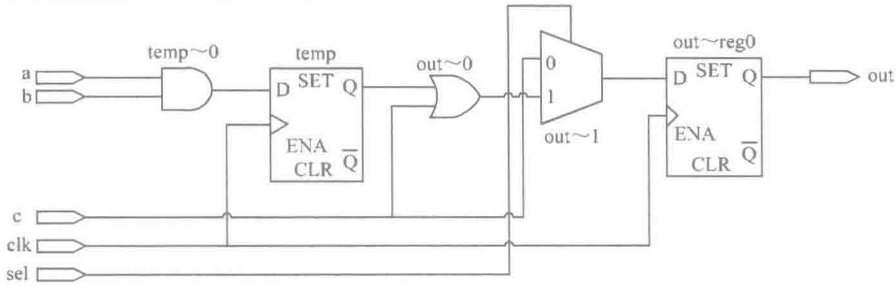


图 3.2-7 程序(2)的电路结构

3.2.4 过程连续赋值语句

过程连续赋值可以在 always 和 initial 过程语句中对连线型和寄存器型变量类型进行赋值操作。在 Verilog HDL 中，过程连续赋值语句有两种类型：赋值、重新赋值语句(assign、deassign)和强制、释放语句(force、release)。值得注意的是，过程连续赋值不能够对寄存器型变量进行位操作，例如，“assign c[1] = 1;”语句将会出现错误。

1. 赋值语句和重新赋值语句

赋值语句和重新赋值语句采用的关键字是“assign”和“deassign”，语法格式分别是：

assign <寄存器型变量> = <赋值表达式>;

和

deassign <寄存器型变量>;

赋值语句只能用于对寄存器型变量赋值，而不可用于对连线型变量赋值；重新赋值语句用于释放 assign 对寄存器型变量的连续赋值，作用后，该寄存器变量仍将保持 deassign 语句执行前的原有取值。也就是说，使用 assign 给寄存器型变量赋值之后，该值将一直保持在这个寄存器上，直至遇到 deassign。

例 3.2-9 使用 assign 和 deassign 设计异步清零 D 触发器。

```
module assign_dff(d, clr, clk, q);
    input d, clr, clk;
    output q;
    reg q;
    always@(clr)
        begin
            if(!clr)
                assign q = 0;                  //时钟沿到来时，d 的变化对 q 无效
            else
                deassign q;
        end
    always@(negedge clk) q = d;
endmodule
```

该例中，如果 clr 为 0，则 assign 赋值语句使 q 清 0。此时，不管时钟和 d 如何变化，

对 q 都没有影响。如果 clr 变为 1，deassign 重新赋值语句被执行，这就使得 assign 强制赋值方式被取消，以后 clk 将能够对 q 产生影响。deassign 语句是一条撤销过程连续赋值的语句。执行后，原来由 assign 语句对变量进行的过程连续赋值操作将失效，寄存器变量被过程连续赋值的状态将解除。

2. 强制语句和释放语句

强制语句和释放语句采用的关键字是“force”和“release”，可以对连线型和寄存器型变量进行赋值操作，force 语句的优先级高于 assign 语句。其语法格式分别是：

force <寄存器或连线型变量> = <赋值表达式>;

和

release<寄存器或连线型变量>;

当 force 语句对寄存器型变量赋值时，变量的当前值被 force 覆盖，因而限制了其它驱动源的作用，直至遇到 release(释放语句，作用类似于 deassign)语句，变量才得以释放，被重新赋值。这种语句主要用于 Verilog HDL 仿真测试程序中，便于对某种信号进行临时性的赋值和测试。

例 3.2-10 force 和 release 使用示例。

```
module force_release(a, b, out);
  input a, b;
  output out;
  wire out;
  and (out, a, b);
  initial
           begin
             force out = alb;
             #5;
             release out;
             end
endmodule
module release_tb;
  reg a, b;
  wire out;
  force_release U1(a, b, out);
  initial
     begin
         a = 1; b = 0;
     end
endmodule
```

例中 force 语句是对连线型变量 out 进行赋值操作。在执行测试模块时，在 0 时刻，主模块中的门级模块(两输入与门)的引用语句和 initial 语句同时执行，所以 force 语句强制生

效，因此，out 的值为 1(out = a | b)，而不是 0(out = a&b)；在第 5 时间单位时刻，由于执行 release 语句，因此中止了 force 语句的连续赋值，此时恢复了门级模块的引用，out 的值为 0(out = a&b)。

3.2.5　条件分支语句

Verilog HDL 的条件分支语句有两种：if 语句和 case 语句。

1. if 语句

if 语句就是判断所给的条件是否满足，然后根据判断的结果来确定下一步的操作。条件语句只能在 initial 和 always 语句引导的语句块(begin-end)中使用，模块的其它部分都不能使用。if 语句有三种形式：

形式 1：

```
if(条件表达式)语句块;
```

形式 2：

```
if(条件表达式)
   语句块 1;
else
   语句块 2;
```

形式 3：

```
if(条件表达式 1)
   语句块 1;
else if(条件表达式 2)
   语句块 2;
   …
else if(条件表达式 i)
     语句块 i;
else
     语句块 n;
```

形式 1 中，当条件表达式成立(逻辑值为 1)时，执行后面的语句块；当条件表达式不成立时，后面的语句块不被执行。例如：

```
if(a>b) out = din;
```

表示当 a > b 时，out 为 din。

形式 2 中，当条件表达式成立时，执行后面的语句块 1，然后结束条件语句的执行；当条件表达式不成立时，执行 else 后面的语句块 2，然后结束条件语句的执行。

例 3.2-11　if-else 使用示例(1)。

```
module mux2_1(a, b, sel, out);
   input a, b, sel;
   output out;
   reg out;
   always@(a, b, sel)
      begin
         if(sel)    out = a;
         else       out = b;
      end
endmodule
```

当 sel 为真(1)时，输出端 out 得到 a 的值；当 sel 为假(0)时，输出端 out 得到 b 的值。这是一个典型的 2 选 1 数据选择器。

形式 3 是多路选择控制，执行的过程是：首先判断条件表达式 1，若为真则执行语句块 1，若为假则继续判断条件表达式 2，然后再选择是否执行语句块 2，依此类推。从条件表达式 1 到条件表达式 n 的排列顺序，可以看出这种形式的条件语句是分先后次序的，本身隐含着一种优先级关系。在实际使用中，有时就需要利用这一特性来实现优先级控制，但有时则要注意避免它给不需要优先级的电路设计带来的影响。

例 3.2-12 if-else 使用示例(2)。

```
module compare_a_b(a, b, out);
   input a, b;
   output [1:0]out;
   reg [1:0]out;
   always@(a, b)
      begin
          if(a>b)      out = 2'b01;
          else if(a == b)     out = 2'b10;
          else   out = 2'b11;
      end
endmodule
```

该例中，首先判断 a 是否大于 b，然后判断 a 是否等于 b，蕴含了优先级的特性。这种特性会在综合后的电路中体现出来。

在 if 语句中允许一个或多个 if 语句的嵌套使用，其语法格式是：

```
if(条件表达式 1)
      if(条件表达式 2)           //内嵌的 if 语句
          语句块 1;
      else
         语句块 2;
else
      if(条件表达式 3)           //内嵌的 if 语句
          语句块 3;
      else
          语句块 4;
```

注意，三种形式的 if 语句在 if 后面都有“表达式”，一般为逻辑表达式或关系表达式。系统对表达式的值进行判断，若为 0、x、z，则按“假”处理；若为 1，则按“真”处理，执行指定的语句块。例如：

```
if(a)       等价于   if(a == 1)
if(!a)      等价于   if(a != 1)
```

2. case 语句

相对于 if 语句只有两个分支而言，case 语句是一种可实现多路分支选择控制的语句，比 if-else 语句更为方便和直观。一般的，case 语句多用于多条件译码电路的设计，如描述

译码器、数据选择器、状态机及微处理器的指令译码等。case 语句的语法格式是：

```
case(控制表达式)
        值 1：语句块 1
        值 2：语句块 2
        ...
        值 n：语句块 n
        default：语句块 n+1
endcase
```

case 语句的执行过程是：当 case 语句中控制表达式的值与值 1 相同时，执行语句块 1；当控制表达式的值与值 2 相同时，执行语句块 2；依此类推。如果控制表达式的值与上面列出的值 1 到值 n 都不相同，则执行 default 后面的语句块 n + 1。

当用 case 语句对控制表达式和其后的值进行比较时，必须是一种全等比较，必须保证两者的对应位全等。case 语句的真值表如表 3.2-3 所示。

表 3.2-3　case 语句的真值表

case	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

注意：

(1) 值 1 到值 n 必须各不相同，一旦判断到与某值相同并执行相应语句块后，case 语句的执行便结束。

(2) 如果某几个连续排列的值项执行的是同一条语句，则这几个值项间可用逗号间隔，而将语句放在这几个值项的最后一项中。

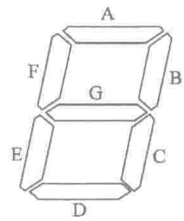
(3) default 选项相当于 if-else 语句中的 else 部分，可依据需要使用或者不使用，当前面已经列出了控制表达式的所有可能值时，default 可以省略。

(4) case 语句的所有表达式的值的位宽必须相等，因为只有这样，控制表达式和分支表达式才能进行对应位的比较。

例 3.2-13　用 case 语句描述 BCD 数码管译码。

```
module BCD_decoder(out, in);
   output[6:0]out;
   input[3:0]in;
   reg[6:0]out;
   always@(in)
        begin
            case(in)
               4'b0000:out = 7'b1111110;
               4'b0001:out = 7'b0110000;
```

```
            4'b0010:out = 7'b1101101;
            4'b0011:out = 7'b1111001;
            4'b0100:out = 7'b0110011;
            4'b0101:out = 7'b1011011;
            4'b0110:out = 7'b1011111;
            4'b0111:out = 7'b1110000;
            4'b1000:out = 7'b1111111;
            4'b1001:out = 7'b1111011;
            default:out = 7'bx;
        endcase
    end
endmodule
```



(a)

字形	输 入	输 出
	in	ABCDEFG
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1111011

(b)

图 3.2-8　BCD 数码管及其真值表

BCD 数码管及其真值表如图 3.2-8 所示。

在使用 case 语句时，应包含所有的状态，如果未包含完全，那么缺省项必须写出，否则将产生锁存器，这在同步时序电路设计中是不允许的。

例 3.2-14　case 语句使用示例。

程序(1)：会产生锁存器的 case 语句。

```
module latch_case(a, b, sel, out);
  input a, b;
  input [1:0]sel;
    output out;
    reg out;
    always@(a, b, sel)
      case(sel)
          2'b00:out = a;
          2'b11:out = b;
      endcase
 endmodule
```

程序(2)：不会产生锁存器的 case 语句。

```
module non_latch_case(a, b, sel, out);
   input a, b;
   input [1:0]sel;
   output out;
   reg out;
   always@(a, b, sel)
        case(sel)
            2'b00:out = a;
            2'b11:out = b;
            default:out = 0;
```

```
    endcase
endmodule
```

除了 case 语句以外，还有 casez、casex 这两种功能类似的条件分支语句，相应的真值表如表 3.2-4 所示。

表 3.2-4(a)　casez 语句的真值表

casez	0	1	x	z
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z	1	1	1	1

表 3.2-4(b)　casex 语句的真值表

casex	0	1	x	z
0	1	0	1	1
1	0	1	1	1
x	1	1	1	1
z	1	1	1	1

casez 与 casex 语句是 case 语句的两种特殊形式，三者的表示形式完全相同，唯一的差别是三个关键词 case、casez、casex 的不同。在 casez 语句中，如果比较的双方(控制表达式与值项)有一边的某一位的值是 z，那么这一位的比较就不予考虑，即认为这一位的比较结果永远是真，因此只需关注其它位的比较结果。而在 casex 语句中，则把这种处理方式进一步扩展到对 x 的处理，即如果比较的双方(控制表达式与值项)有一边的某一位的值是 z 或 x，那么这一位的比较就不予考虑。

3.2.6　循环语句

Verilog HDL 中规定了四种循环语句，分别是 forever、repeat、while 和 for 语句。与条件分支语句一样，循环语句也是一种高级程序语句，多用于测试仿真程序设计。

1．forever 语句

关键字“forever”所引导的循环语句表示永久循环。在永久循环中不包含任何条件表达式，只执行无限循环，直至遇到系统任务$finish。如果需要从 forever 循环中退出，则可以使用 disable 语句。forever 语句的语法格式是：

```
forever 语句或语句块;
```

forever 语句连续不断地执行后面的语句或语句块，常用来产生周期性的波形，作为仿真激励信号。它与 always 语句的不同之处在于不能独立写在程序中。forever 语句一般用在 initial 过程语句中，如果在 forever 语句中没有加入时延控制，forever 语句将在 0 时延后无限循环下去。

例 3.2-15　用 forever 语句产生时钟信号。

```
module forever_tb;
    reg clock;
    initial
        begin
            clock = 0;
            forever #50clock = ~clock;
        end
endmodule
```

2. repeat 语句

关键字“repeat”所引导的循环语句表示执行固定次数的循环。其语法格式是：

```
repeat(循环次数表达式)
  语句或语句块(循环体);
```

其中，“循环次数表达式”用于指定循环次数，它必须是一个常数、一个变量或者一个信号。如果循环次数是变量或者信号，则循环次数是循环开始执行时变量或者信号的值，而不是循环执行期间的值。

repeat 语句的执行过程为：先计算出循环次数表达式的值，并将它作为循环次数保存起来；接着执行后面的语句块(循环体)，语句块执行结束后，将重复执行次数减去一次，再接着重新执行下一次的语句块操作，如此重复，直至循环执行次数被减为 0 时，结束整个循环过程。

例 3.2-16 使用 repeat 语句产生固定周期数的时钟信号。

```
module repeat_tb;
    reg clock;
    initial
        begin
            clock = 0;
            repeat(8)    clock = ~clock;
        end
endmodule
```

例中，循环体所预先制定的循环次数为 8 次，相应产生 4 个时钟周期信号。

3. while 语句

关键字“while”所引导的循环语句表示的是一种“条件循环”。while 语句根据条件表达式的真假来确定循环体的执行，当指定的条件表达式取值为真时才会重复执行循环体，否则就不执行循环体。其语法格式是：

```
while(条件表达式)  语句或语句块;
```

其中，“条件表达式”表示循环体得以继续重复执行时必须满足的条件，它常常是一个逻辑表达式。在每一次执行循环体之前都要对这个条件表达式是否成立进行判断。

while 语句的执行过程可以描述为：先判断条件表达式是否为真，如果是，则执行后面的语句，接着再回来判断条件表达式是否仍为真，只要是真，再执行语句，直至某一次执行完语句后，判断出条件表达式的值为非真时，结束循环过程。为保证循环过程的正常结束，通常在循环体内部必定有一条语句用以改变条件表达式的值。

例 3.2-17 使用 while 语句产生时钟信号。

```
module while_tb;
    reg clock;
    initial
        begin
            clock = 0;
```

```
            while(1)
            #50  clock = ~clock;
        end
endmodule
```

4. for 语句

关键字“for”所引导的循环语句也表示一种“条件循环”，只有在指定的条件表达式成立时才进行循环。其语法格式是：

for(循环变量赋初值；循环结束条件；循环变量增值)　　语句块；

for 语句的执行过程是：先给“循环变量赋初值”，然后判断“循环结束条件”，若其值为真，则执行 for 语句中指定的语句块，然后进行“循环变量增值”操作，这一过程进行到循环结束条件满足时，for 循环语句结束。

例 3.2-18　使用 for 语句产生时钟信号。

```
module for_clk;
    reg clk;
    integer i;
    initial
        begin
            clk = 0;
            for(i = 0; i >= 0; i = i+1)
            #50 clk = ~clk;
        end
endmodule
```

应该说明的是，循环语句也可以用于可综合电路的设计，当采用循环语句进行计算和赋值的描述时，可以综合得到逻辑电路。

例 3.2-19　用 Verilog HDL 设计一个 8 位移位寄存器。

程序(1)：采用赋值语句实现。

```
module shift_regist1(Q, D, rst, clk);
  output [7:0] Q;
  input D, rst, clk;
  reg [7:0] Q;
  always @(posedge clk)
      if (!rst)    Q <= 8'b000000;
      else     Q <= {Q[6:0], D};
endmodule
```

程序(2)：采用 for 语句实现。

```
module shift_regist2(Q, D, rst, clk);
    output [7:0] Q;
    input D, rst, clk;
    reg [7:0] Q;
```

```
    integer i;
    always @(posedge clk)
      if (!rst) Q <= 8'b000000;
      else
      begin
            for (i = 7; i > 0; i = i-1)      Q[i] <= Q[i-1];
            Q[0] <= D;
      end
endmodule
```

这两种描述方式本是同样的描述形式，因此产生的电路是完全相同的，综合后的结果如图 3.2-9 所示。

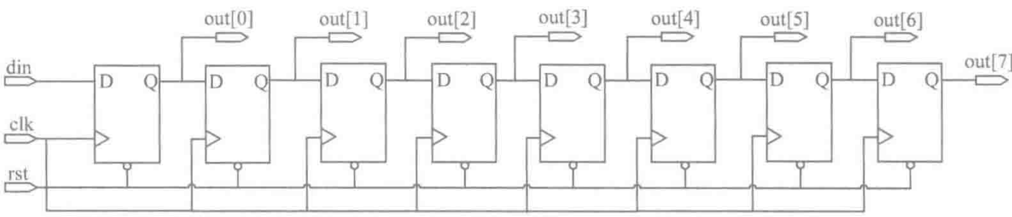


图 3.2-9 例 3.2-19 综合后的结果

3.3 结构化建模

结构描述方式就是将硬件电路描述成一个分级子模块系统，通过逐层调用这些子模块构成功能复杂的数字逻辑电路和系统的一种描述方式。在这种描述方式下，组成硬件电路的各个子模块之间的相互层次关系以及相互连接关系都需要得到说明。由于任何硬件电路在结构上都是由一级级不同层次的若干功能单元组成的，所以结构描述方式很适合用来对电路的结构特点进行说明，这也是“结构描述方式”这种叫法的由来。结构描述方式的描述目标是电路的层次结构，组成硬件电路的各层功能单元将被描述成各个级别的子模块。

根据所调用子模块的不同抽象级别，可以将模块的结构描述方式分成如下三类：

(1) 模块级建模：通过调用由用户设计生成的低级子模块来对硬件电路结构进行说明。这种情况下的模块由低级模块的实例组成。

(2) 门级建模：通过调用 Verilog HDL 内部的基本门级元件来对硬件电路的结构进行说明。这种情况下的模块由基本门级元件的实例组成。

(3) 开关级建模：通过调用 Verilog HDL 内部的基本开关元件来对硬件电路的结构进行说明。这种情况下的模块由基本开关级元件的实例组成。

3.3.1 模块级建模

模块级建模就是通过调用由用户自己描述产生的 module 模块来对硬件电路结构进行说明，并设计出电路。上一章已经对模块的概念进行了说明，下面主要介绍如何调用模块进行模块级建模。

模块级建模方式可以把一个模块看做由其它模块像积木一样搭建而成的。模块中被调用模块属于低一层次的模块。如果当前模块不再被其它模块所调用，那么这个模块一定是所谓的顶层模块。在对一个硬件系统的描述中，必定有而且只能有一个顶层模块。

1. 模块调用方式

在 Verilog HDL 中，模块可以被任何其它模块调用，这种调用实际上是将模块所描述的电路复制并连接。一个模块可以调用多个模块，这些模块可以是相同的，也可以是不同的，语法要求在于同一模块中被调用模块的实例名不同。模块调用的基本语法格式是：

模块名<参数值列表> 实例名(端口名列表);

其中，“模块名”是在 module 定义中给定的模块名，它指明了被调用的是哪一个模块；“参数值列表”是可选项，它是将参数值传递给被调用模块实例中的各个参数；“实例名”是模块被调用到当前模块的标志，用来索引层次化模块建模中被调用模块的位置；“端口名列表”是被调用模块实例各端口相连的外部信号。

例 3.3-1 简单的模块调用示例。

```
module and_2(a, b, c);              //2 输入与门模块
    input a, b;
    output c;
        assign c = a&b;
endmodule
module logic(in1, in2, q);          //顶层模块
    input in1, in2;
    output q;
        and_2    U1(in1, in2, q);   //模块的调用
endmodule
```

该例采用模块调用的方式实现了简单的逻辑运算，包括两个模块。其中第一个 and_2 是自定义的 2 输入与门模块，为底层模块；而第二个模块 logic 是顶层模块，用来调用 and_2 模块。其中“and_2 U1(in1, in2, q);”是模块实例语句，实现对 2 输入模块的调用，采用的是端口的位置对应方式。其中 in1、in2、q 分别与 2 输入与门模块中的 a、b、c 相连接。

如果同一个模块在当前模块中被调用几次，则需要用不同的实例名加以标识，但可在同一条模块调用语句中被定义，只要各自的实例名和端口名列表相互间用逗号隔开即可，基本语法格式如下：

```
模块名    <参数值列表> 实例名 1(端口名列表 1),
          <参数值列表> 实例名 2(端口名列表 2),
          ...
          <参数值列表> 实例名 n(端口名列表 n);
```

在上面的格式当中，“模块名”就是被调用的模块，“参数值列表”是可选项，“实例名”代表生成的模块实例，实例名必须各不相同，“端口名列表”指明了模块实例与外部信号的连接。上面的整个实例语句实现了对模块的 n 次调用，将生成 n 个模块实例。

如在例 3.3-1 中，如果想调用更多的 2 输入与门实现更复杂的功能，可以使用如下的多

条模块实例语句：

```
and_2  U1(a1, b1, out1),
       U2(a2, b2, out2),
       ...
       Un(an, bn, outn);
```

当需要对同一个模块进行多次调用时，还可以采用阵列调用的方式对模块进行调用。阵列调用的语法格式如下：

```
<被调用模块名><实例阵列名>[阵列左边界:阵列右边界](<端口连接表>);
```

其中，“阵列左边界”和“阵列右边界”是两个常量表达式，用来指定调用后生成的模块实例阵列的大小。

例 3.3-2　使用阵列调用方式的模块实例语句来进行结构描述。

```
module AND(andout, ina, inb);            //基本的与门模块
      input ina, inb;
      output andout;
            assign andout = ina&inb;
endmodule
module   ex_arrey(out, a, b);          //顶层模块，用来调用与门模块
     input[15:0] a, b;
     output [15:0] out;
     wire [15:0] out;
          AND    AND_ARREY[15:0](out, a, b);
endmodule
```

例中，“ex_arrey”模块是一个结构描述模块，该模块对“AND”子模块通过阵列调用方式进行调用。其中[15:0]定义了实例阵列的大小，它指明该实例阵列包括 16 个实例的调用。其中模块调用语句等价于如下几条语句：

```
AND    AND_ARREY15(out[15], a[15], b[15]);
…
AND    AND_ARREY1(out[1], a[1], b[1]);
AND    AND_ARREY0(out[0], a[0], b[0]);
```

可以看出，通过采用阵列调用方式，可以极大地简化程序，节约资源，使程序结构一目了然。

2. 模块端口对应方式

在模块级建模方式中，被调用模块需要将模块的输入和输出信号连接到调用模块中。在 Verilog HDL 中有两种模块调用端口对应方式，即端口位置对应方式和端口名对应方式。

1) 端口位置对应方式

端口位置对应方式是被调用的模块按照一定的顺序出现在端口连接表中的一种模块调用方式，其语法格式如下：

模块名<参数值列表> 实例名(<信号名 1>，<信号名 2>，…，<信号名 n>);

其中，端口名列表中的这些信号将与所调用模块定义的端口依次连接，如信号名 1 与所调用模块端口列表中的第一个端口相连接，信号名 2 与所调用端口列表中的第二个端口相连接，依此类推。

例 3.3-3　采用模块结构建模方式用 1 bit 半加器构成 1 bit 全加器。

```
module halfadder(a, b, s, c);          //半加器模块
    input a, b;
      output c, s;
        assign s = a^b;
        assign c = a&b;
endmodule
module fulladder(p, q, ci, co, sum); //全加器模块
    input p, q, ci;
    output co, sum;
        wire w1, w2, w3;
        halfadder   U1(p, q, w1, w2);
        halfadder   U2(ci, w1, sum, w3);
        or U3(co, w2, w3);
endmodule
```

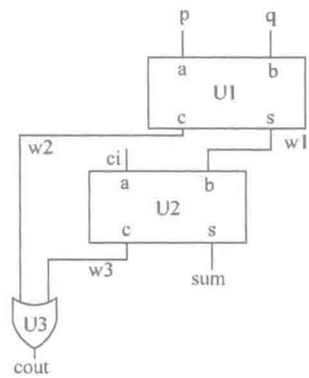


图 3.3-1　由 1 bit 半加器构成的 1 bit 全加器电路

其模块化电路结构如图 3.3-1 所示。

在模块实例引用语句中，halfadder 是模块的名称，U1、U2 是实例名称，并且端口是按照位置对应关联的。在第一个模块实例引用中，信号 p 与模块 halfadder 的端口 a 连接，信号 q 与端口 b 连接，信号 w1 与端口 s 连接，信号 w2 与端口 c 连接。第二个模块实例引用中的对应关系与第一个模块实例相似。

若端口列表中的某一项的“信号名”缺省，则表示这一项所对应的模块端口未被连接(悬空)。在实际电路设计过程中，常常需要将不需要使用的端口悬空，可通过缺省该信号名来实现。比如将上例中的模块实例语句改为以下的模块实例语句：

```
halfadder   U1(p, q, , w2);
```

端口列表中第三项的信号名是缺省的，这种情况表明该项对应的模块端口(端口列表中的第三个端口 s)是悬空的。

2) 端口名对应方式

端口名对应方式是 Verilog HDL 允许的另一种模块调用方式，其语法格式如下：

模块名 <参数值列表> 实例名(.端口名 1<信号名 1>，.端口名 2<信号名 2>，…，.端口名 n<信号名 n>);

在这种方式中，模块定义时的端口名和调用时的实际连接信号名之间的一一对应关系被显式地表示出来。由于端口之间的对应关系已十分明确，因而这种情况下，调用时端口名的排列顺序可以随意改变。需要注意的是，在模块引用时用“.”标明所调用模块定义的端口名。

例 3.3-4 端口名对应的模块调用示例。

```
module dff(d, clk, clr, q);                //D 触发器模块，是被调用的模块，属于底层模块
    input d, clk, clr;
    output q;
    reg q;
    always@(posedge clk or negedge clr)
        begin
            if(!clr)
                q = 0;
            else
                q = d;
        end
endmodule
module shifter_D(din, clock, clear, out);  //顶层模块，用来调用底层模块
    input din, clock, clear;
    output [3:0]out;
        dff    U1(.q(out[0]), .d(din), .clk(clock), .clr(clear));
        dff    U2(.q(out[1]), .d(out[0]), .clk(clock), .clr(clear));
        dff    U3(.q(out[2]), .d(out[1]), .clk(clock), .clr(clear));
        dff    U4(.q(out[3]), .d(out[2]), .clk(clock), .clr(clear));
endmodule
```

其模块化电路结构如图 3.3-2 所示。

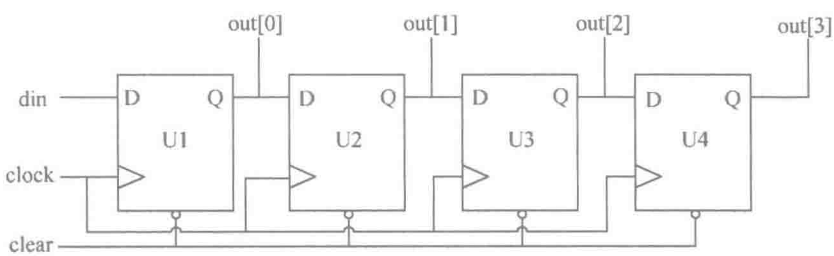


图 3.3-2 例 3.3-4 的模块化电路结构

上例是一个标准的模块级建模的例子，对于模块 dff，是用行为建模定义的一个上升沿异步清零的 D 触发器。在模块 shifter_D 中描述的是一个 4 位的移位寄存器，它是利用 4 次调用模块 dff 实现的，采用了端口名对应方式进行模块的调用。其中，模块 dff 是底层模块，是被调用的模块；相对来说，模块 shifter_D 就是顶层模块，用来调用 dff 模块。

对于第一个模块实例语句“dff U1(.q(out[0]), .d(din), .clk(clock), .clr(clear));”，被调用模块名是“dff”；实例名是“U1”；端口连接表是“(.q(out[0]), .d(din), .clk(clock), .clr(clear))”，它表示模块实例 U1 的各个端口将分别与模块 dff 中的变量 q、d、clk、clr 相连接，与它们

的顺序没有关系。后面几个模块实例语句与此类似。

在端口名对应方式下，如果端口连接表内某一项的信号端口是缺省的(在“信号名”位置处只出现一对空括号“()”)，那么这一项的端口名所代表的端口处于悬空状态。比如将上例中的模块实例语句改为下面的模块实例语句：

```
dff U4(.q(), .d(out[2]), .clk(clock), .clr(clear));
```

端口列表中第一项内的信号名是缺省的(空括号)，表明端口“q”悬空。

3) 不同端口位宽的匹配

在端口和端口表达式之间存在着一种隐含的连续赋值语句，因此当端口和端口表达式的位宽不一致时，会进行端口匹配，其采用的位宽匹配规则与连续赋值时使用的规则相同。

例 3.3-5 模块调用时不同位宽的匹配问题。

```
module ex1(a, b);
    input [6:1]a;
    output [3:0]b;
    ...
endmodule
module test;
    wire[5:3]c;
    wire[5:1]d;
        ex1 U1(.a(c), .b(d));
            ...
endmodule
```

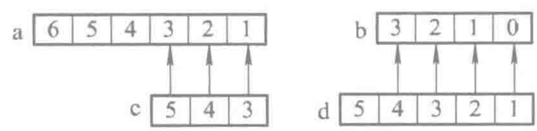


图 3.3-3 端口匹配

上例中，c[3]、c[4]、c[5]分别与 a[1]、a[2]、a[3]相连接，输入端口 a 剩余的位没有连接。同样，d[1]、d[2]、d[3]、d[4]分别接到输出端口 b 的 b[0]、b[1]、b[2]、b[3]上。其连接对应关系见图 3.3-3。

3. 模块参数值

当一个模块在另一个模块的内部被实例引用时，较高层次的模块能够通过调用较低层次的模块来改变低层次模块的值。这个功能在实际的设计当中很有用，能够降低程序的复杂度，使程序简化。比如设计加法器时，加法器的位数可以由模块内的参数来确定。在调用这个加法器模块时，只需修改参数的值就会得到不同位数的加法器，非常方便。

通过下面两种途径可以改变模块实例的参数值。

1) 使用带有参数的模块实例语句修改参数值

在这种方法中，模块实例本身就能指定新的参数值。其语法格式如下：

```
模块名<参数值列表>调用名(端口名列表);
```

其中，“参数值列表”又分为位置对应和名称对应两种方式。

例 3.3-6 模块调用改变参数值示例。

```
module para1(C, D);
```

```
        parameter a = 1;
        parameter b = 1;
        ...
    endmodule
    module para2;
        ...
        para1   #(4, 3)   U1(C1, D1);                    //语句 1
        para1   #(.b(6), .a(5))   U2(C2, D2);            //语句 2
        ...
    endmodule
```

该例通过在模块 para2 中对模块 para1 的引用，改变了模块 para1 中的参数值。其中语句 1 是利用位置对应方式将 4 传给 a、3 传给 b，这和模块 para1 中定义参数的先后顺序有关；而语句 2 则是利用名称对应方式将 6 传给 b、5 传给 a，此时和模块 para1 中定义参数的顺序无关。

2) 使用参数重定义语句 defparam 修改参数值

参数重定义语句 defparam 的语法格式如下：

```
defparam    参数名 1 = 参数值 1,
            参数名 2 = 参数值 2,
            ...
            参数名 n = 参数值 n;
```

需要注意的是，“参数名”必须采用分级路径的形式，才能锁定需要修改的参数。

例 3.3-7 使用 defparam 语句修改参数值。

```
module halfadder(a, b, s, c);                 //半加器模块 halfadder
    input a, b;
    output c, s;
    parameter xor_delay = 2, and_delay = 3;
      assign #xor_delay s = a^b;
      assign #and_delay c = a&b;
endmodule
module fulladder(p, q, ci, co, sum);          //全加器模块 fulladder
    input p, q, ci;
    output co, sum;
    parameter or_delay = 1;
      wire w1, w2, w3;
          halfadder      U1(p, q, w1, w2);
          halfadder      U2(ci, w1, sum, w3);
          or #or_delay   U3(co, w2, w3);
endmodule
```

```
module top1(top1a, top1b, top1s, top1c);  //修改半加器模块参数的模块 top1
   input top1a, top1b;
   output top1s, top1c;
       defparam        U1.xor_delay = 4,
                       U1.and_delay = 5;
   //名为 U1 的半加器实例中对参数 xor_delay 和参数 and_delay 值的修改
   halfadder U1(top1a, top1b, top1s, top1c);
endmodule
module top2(top2p, top2q, top2ci, top2co, top2sum);  //修改全加器模块参数的模块 top2
   input top2p, top2q, top2ci;
   output top2co, top2sum;
         defparam        U2.U1.xor_delay = 6,
                         U2.U2.and_delay = 7;
    //名为 U2 的全加器实例中引用的名为 U3 的半加器实例中对参数 xor_delay 和
    //参数 and_delay 值的修改
           U2.or_delay = 5;
    //名为 U2 的全加器实例中对参数 or_delay 值的修改
    fulladder U2(top2p, top2q, top2ci, top2co, top2sum);
endmodule
```

3.3.2 门级建模

1. Verilog HDL 基本门级元件的类型

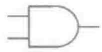
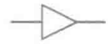
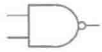
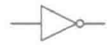
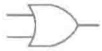
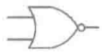
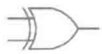
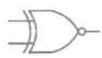
Verilog HDL 中内置有 26 个基本元件，其中 14 个是门级元件，12 个为开关级元件，这些基本元件及其分类见表 3.3-1。

表 3.3-1 Verilog HDL 中内置基本元件及其分类

类型		元件
基本门	多输入门	and, nand, or, nor, xor, xnor
	多输出门	buf, not
三态门	允许定义驱动强度	bufif0, bufif1, notif0, notif1
MOS 开关	无驱动强度	nmos, pmos, cmos rnmos, rpmos, rcmos
双向开关	无驱动强度	tran, tranif0, tranif1 rtran, rtranif0, rtranif1
上拉、下拉电阻	允许定义驱动强度	pullup, pulldown

这里重点介绍门级元件。Verilog HDL 中丰富的门级元件为电路的门级结构提供了方便。Verilog HDL 中常用的内置门级元件见表 3.3-2。

表 3.3-2 Verilog HDL 中常用的内置门级元件

类别	关键字	符号示意图	门名称	类别	关键字	符号示意图	门名称
多输入门	and		与门	多输出门	buf		缓冲器
	nand		与非门		not		非门
	or		或门	三态门	bufif1		四种三态门
	nor		或非门		bufif0		
	xor		异或门		notif1		
	xnor		异或非门		notif0		

2. 门级模块调用

多输入门元件调用的语法格式如下：

元件名<实例名>(<输出端口>，<输入端口 1>，<输入端口 2>，…，<输入端口 n>)；

例如：

```
and   A1 (out1, in1, in2);
or    O2 (a, b, c, d);
xor   X1(x_out, p1, p2);
```

多输出门元件调用的语法格式如下：

元件名<实例名>(<输出端口 1>，<输出端口 2>，…，<输出端口 n>，<输入端口>)；

例如：

```
not   NOT_1 (out1, out2, in);
buf   BUF_1 (bufout1, bufout2, bufout3, bufin);
```

三态门元件调用的语法格式如下：

元件名<实例名>(<数据输出端口>，<数据输入端口>，<控制输入端口>)；

例如：

```
bufif1  BF1 (data_bus, mem_data, enable);
bufif0  BF0 (a, b, c);
notif1  NT1 (out, in, ctrl);
notif0  NT0 (addr, a_bus, select);
```

例 3.3-8 调用门级元件实现图 3.3-4 所示的 2 线-4 线译码器。

```
module decoder2_4(in0, in1, en, out0, out1, out2, out3);
    output out0, out1, out2, out3;
    input in0, in1, en;
        wire wire1, wire2;
```

```
        not     U1(wire1, in0),
                U2(wire2, in1);
        nand    U3(out0, en, wire1, wire2),
                U4(out1, en, wire1, in1),
                U5(out2, en, in0, wire2),
                U6(out3, en, in0, in1);
endmodule
```

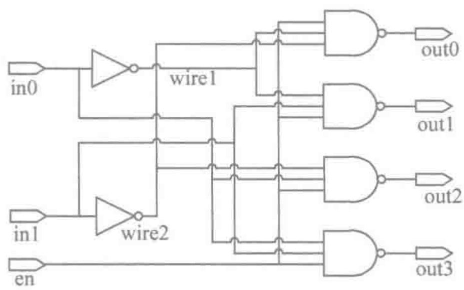


图 3.3-4　用基本门实现的 2 线-4 线译码器

3.3.3 开关级建模

开关级建模指的是用输入、输出为模拟信号的晶体管搭建硬件模型。Verilog HDL 提供了开关级建模方式，主要用于 ASIC 设计。在开关级建模方式下，硬件结构采用开关级描述方式，而晶体管的输入、输出均被限定为数字信号。此时，晶体管表现为通断形式的开关。由于 Verilog HDL 采用四值逻辑系统，因此 Verilog HDL 描述的开关的输入、输出可以为 0、1、z 或 x。

Verilog HDL 提供了十几种开关级基本元件，它们是实际的 MOS 管的抽象表示。这些开关级基本元件分为两大类：一类是 MOS 开关，一类是双向开关。每一大类又可分为电阻型(前缀用 r 表示)和非电阻型。本节主要以非电阻型开关为例，介绍 MOS 开关和双向开关。

1) MOS 开关

MOS 开关模拟了实际的 MOS 器件的功能，包括 nmos、pmos、cmos 三种。

nmos 和 pmos 开关的实例化语言格式如下：

```
nmos/pmos  实例名(out，data，control);
```

coms 开关的实例化语言格式如下：

```
cmos  实例名(out，data，ncontrol，pcontrol);
```

2) 双向开关

MOS 开关只提供了信号的单向驱动能力，为了模拟实际的具有双向驱动能力的门级开关，Verilog HDL 提供了双向开关。双向开关的每个脚都被声明为 inout 类型，可以作为输入驱动另一脚，也可以作为输出被另一脚驱动。

双向开关包括无条件双向开关(tran)和有条件双向开关(tranif0、tranif1)。

无条件双向开关的实例化语言格式如下：

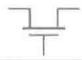
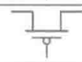
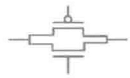
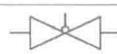
```
tran  实例名  (inout1，inout2);
```

有条件双向开关的实例化语言格式如下：

tranif0/tranif1 实例名(inout1，inout2，control);

表 3.3-3 列出了 Verilog HDL 提供的开关级元件。

表 3.3-3 Verilog HDL 提供的开关级元件

开关类别	关键字	符号示意图	功能说明
MOS开关	nmos		当控制信号为高时，开关导通，否则关闭
	pmos		当控制信号为低时，开关导通，否则关闭
	cmos		nmos控制信号和pmos控制信号互补，当nmos控制信号为高时，开关导通，否则关闭
双向开关	tran		两端可以互相驱动，且随时保持一致
	tranif0		当控制端为低时，两端才能互相驱动
	tranif1		当控制端为高时，两端才能互相驱动

例 3.3-9 基本的 nmos 开关电路示例。

```
module aNMOS(din, ctr, out);
  input din, ctr;
  output out;
     nmos U1(out, din, ctr);
endmodule
```

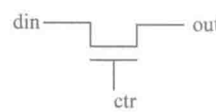


图 3.3-5 nmos 开关电路

nmos 开关电路如图 3.3-5 所示。

当 ctr 为高电平时，输入 din 传到输出端 out；当 ctr 为低电平时，输入、输出断开。这是最基本的 nmos 开关，pmos 开关类似。

例 3.3-10 cmos 开关级 2 输入与非门示例。

2 输入与非门的 cmos 电路如图 3.3-6 所示，其 Verilog HDL 程序代码如下：

```
module nnand2(a, b, y);
     input a, b;
     output y;
     supply0 Gnd;
     supply1 Vdd;   //supply0 和 supply1 为内部参
                    //量，分别表示低电平和高电平
     wire im1;
          pmos g1(y, Vdd, a);
          pmos g2(y, Vdd, b);
          nmos g3(y, im1, a);
          nmos g4(im1, Gnd, b);
endmodule
```

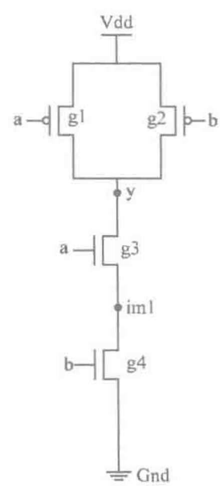


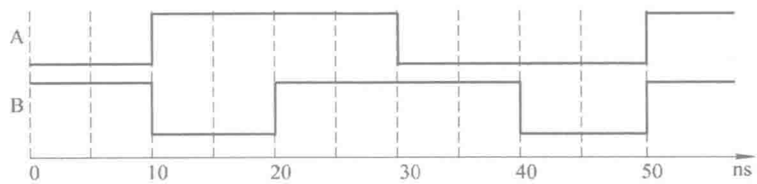
图 3.3-6 2 输入与非门的 cmos 电路

本 章 小 结

本章主要介绍了 Verilog HDL 程序设计语句和描述方式，给出了各种高级程序设计语句的概念和用法，并且详细描述了 Verilog HDL 的三大建模方式(数据流建模、行为级建模、结构化建模)。这些内容是后续复杂数字电路设计的必要基础。

思考题和习题

1. 举例说明连续赋值语句和过程赋值语句的区别。连续赋值语句能否放在过程块里？
2. 试用连续赋值语句描述一个 4 选 1 的数据选择器。
3. Verilog HDL 中常用的复位方式有哪些？如何描述同步和异步复位？
4. 举例说明 always 过程块和 initial 过程块的区别。
5. 简述 begin-end 语句块和 fork-join 语句块的区别，并写出下面信号对应的程序代码。



6. 试分析下面两段代码的区别与联系。

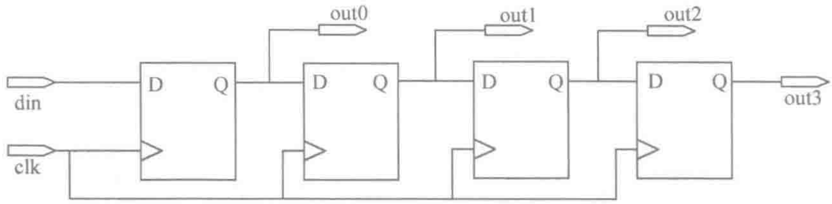
代码 1：

```
always@(posedge clk)
    begin
        for(i = 0; i <= 63; i = i+1)
        mem[i] = i;
    end
```

代码 2：

```
initial
    begin
        for(i = 0; i <= 63; I = i+1)
        begin
            mem[i] = i;
            @(posedge clk)
        end
    end
```

7. 举例说明阻塞赋值语句和非阻塞赋值语句的区别。
8. 分别用阻塞赋值语句和非阻塞赋值语句描述下图所示移位寄存器的电路。



9. 画出以下 Verilog HDL 程序所描述的电路，并分析电路结构。

```
module program_if(a, b, c, d, sel, z);
    input a, b, c, d;
    input [3:0] sel;
    output z;
    reg z;
    always @(a or b or c or d or sel)
        begin
            if (sel[3])          z = d;
            else if (sel[2])     z = c;
            else if (sel[1])     z = b;
            else if (sel[0])     z = a;
            else                 z = 0;
    end
endmodule
```

10. 简述 case、casex 和 casez 之间的异同。

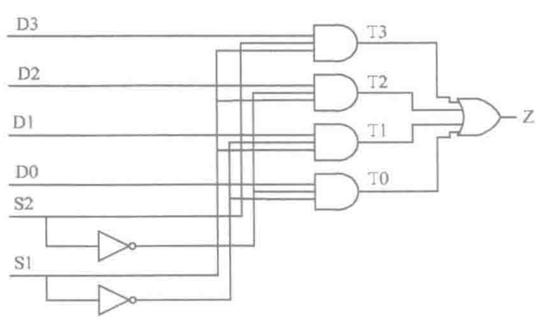
11. 在串行块和并行块中，forever 语句是否影响其后语句的运行？

12. Verilog HDL 模块的结构描述方式有哪三种？

13. 根据下面的 Verilog HDL 程序，画出电路图，并说明它所完成的功能。

```
module circuit(a, b, c);
    input a, b;
    output c;
        wire a1, a2, anot, bnot;
        and U1(a1, a, b);
        and U2(a2, anot, bnot);
          not (anot, a);
        not (bnot, b);
        or (c, a1, a2);
endmodule
```

14. 试用 Verilog HDL 门级描述方式描述右图所示的电路。



第 4 章 Verilog HDL 数字逻辑电路设计方法

4.1 Verilog HDL 的设计思想和可综合特性

在数字集成电路设计过程中，设计者使用 Verilog HDL 进行关键性步骤的开发和设计。其基本过程是，首先使用 Verilog HDL 对硬件电路进行描述性设计，利用 EDA 综合工具将其综合成一个物理电路，然后进行功能验证、定时验证和故障覆盖验证。

与计算机软件所采用的高级程序语言(C 语言)类似，Verilog HDL 是一种高级程序设计语言，程序编写较简单，设计效率很高。然而，它们面向的对象和设计思想却完全不同。

软件高级程序语言是对通用型处理器(如 CPU)的编程，主要是在固定硬件体系结构下的软件化程序设计。处理器的体系结构和功能决定了可以用于编程的固定指令集，设计人员的工作是调用这些指令，在固化的体系结构下实现特定的功能。

Verilog HDL 和 VHDL 等硬件描述语言对电路的设计是将基本的最小数字电路单元(如门单元、寄存器、存储器等)通过连接方式，构成具有特定功能的硬件电路。在数字集成电路中，这种最小的单元是工艺厂商提供的设计标准库或定制单元；在 FPGA 中，这种最小的单元是芯片内部已经布局的基本逻辑单元。设计人员通过描述性语言调用和组合这些基本单元实现特定的功能，其基本的电路是灵活的。

Verilog HDL 给设计者提供了几种描述电路的方法。设计者可以使用结构描述方式把逻辑单元互连在一起进行电路设计，也可以采用抽象描述方式对大规模复杂电路进行设计，如对有限状态机、数字滤波器、总线和接口电路的描述等。

由于硬件电路的设计目标是最终产生的电路，因此 Verilog HDL 程序设计的正确性需要通过对综合后电路的正确性进行验证来实现。逻辑上相同的电路在物理电路中的形式却有可能完全不同。对于 Verilog HDL 程序设计而言，数字电路的描述性设计具有一定的设计模式，这与 C 语言等高级软件程序设计是不同的。

例 4.1-1 是对模 256(8 bit)计数器的两种描述。程序(1)是通常的 Verilog HDL 对计数器的描述方式，通过改变计数器状态寄存器组的位宽和进位条件，可以实现对不同计数器的硬件电路设计。程序(2)是初学者经常出现的一种错误描述方式，刚开始编写 Verilog HDL 程序时经常会套用 C 语言等高级程序设计的模式，这样往往得不到目标数字电路的功能。

例 4.1-1 用 Verilog HDL 设计模 256(8 bit)计数器。

(1) 可综合程序描述方式：

```
module counter (count, clk, reset);
    output count;
```

```
    input clk, reset;
    reg [7:0] count;
    reg out;
        always @(posedge clk)
            if (!reset)     count <= 0;
            else if (count == 8'b11111111)       count <= 0;
            else            count <= count+1;
endmodule
```

(2) 常见的错误描述方式：

```
module counter (count, clk, reset);
    output count;
    input   reset, clk;
    reg [7:0] count;
    reg out;
    integer i;
        always @(posedge clk, reset)
            begin
                if (!reset) count <= 0;
                else
                  for (i = 0; i <= 255; i = i+1)
                        count <= count+1;
            end
endmodule
```

Verilog HDL 的电路描述方式具有多样性，这也决定了电路设计的多样性。例 4.1-2 是对一个多路选择器的设计，程序(1)采用的是真值表的形式，程序(2)采用的是逻辑表达式的形式，程序(3)采用的是基本逻辑单元的结构性描述形式。

例 4.1-2　用 Verilog HDL 设计数字多路选择器。

(1) 采用真值表形式的程序代码：

```
module MUX (out, data, sel);
    output out;
    input [3:0] data;
    input [1:0] sel;
    reg out;
        always @(data or sel)
            case (sel)
              2'b00 : out<=data[0];
              2'b01 : out <= data[1];
              2'b10 : out <= data[2];
              2'b11 : out <= data[3];
```

```
      endcase
   endmodule
```

(2) 采用逻辑表达式形式的程序代码：

```
module MUX (out, data, sel);
     output out;
     input [3:0] data;
     input [1:0] sel;
     wire w1, w2, w3, w4;
        assign w1 = (~sel[1])&(~sel[0])&data[0];
        assign w2 = (~sel[1])&sel[0]&data[1];
        assign w3 = sel[1]&(~sel[0])&data[2];
        assign w4 = sel[1]&sel[0]&data[3];
        assign out = w1|w2|w3|w4;
endmodule
```

(3) 采用结构性描述形式的程序代码：

```
module MUX (out, data, sel);
     output out;
     input [3:0] data;
     input [1:0] sel;
     wire w1, w2, w3, w4, w5, w6;
       not    U1 (w1, sel[1]);
              U2 (w2, sel[0]);
       and    U3 (w3, w1, w2, data[0]);
              U4 (w4, w1, sel[0], data[1]);
              U5 (w5, sel[1], w2, data[2]);
              U6 (w6, sel[1], sel[0], data[3]);
       or     U7 (out, w3, w4, w5, w6);
endmodule
```

Verilog HDL 主要用于电路设计和验证，部分语言是为电路的测试和仿真制定的，因此其语言分为用于电路设计的可综合性语言和用于测试仿真的不可综合性语言。对于可综合性语言，EDA 综合工具可以将其综合为物理电路，而对于其中的部分语言，EDA 工具的综合性很差，设计人员往往得不到与设计思想相符合的物理电路。

正是由于 Verilog HDL 的特殊性，初学者往往很难把握可综合电路的设计方法，得不到最终期望的电路，这也是采用 Verilog HDL 进行设计所面临的一个困难。为了解决这一问题，降低 Verilog HDL 的设计门槛，EDA 工具厂商正努力设计综合性工具，使综合性工具能够适应 C 语言程序设计思想，但这需要一个很长的过程。

在现阶段，作为设计人员，熟练掌握 Verilog HDL 程序设计的多样性和可综合性是至关重要的。作为数字集成电路的基础，基本数字逻辑电路的设计是进行复杂电路设计的前提。本章通过数字电路中基本逻辑电路的 Verilog HDL 程序设计进行介绍，要求读者掌握基本逻

辑电路的可综合性设计，为具有特定功能的复杂电路的设计打下基础。

逻辑电路可以分成两大类：一类是组合逻辑电路，简称组合电路；另一类是时序逻辑电路，简称时序电路。本章将分别从这两种电路的原理和 Verilog HDL 程序设计方法出发，对数字逻辑电路的基本功能电路进行设计，这也是复杂数字集成电路系统设计的基础。

4.2 组合电路的设计

组合电路的特点是，电路中任意时刻的稳态输出仅仅取决于该时刻的输入，而与电路原来的状态无关。组合电路没有记忆功能，只有从输入到输出的通路，没有从输出到输入的回路。

组合电路的设计需要从以下几个方面考虑：首先，所用的逻辑器件数目最少，器件的种类最少，且器件之间的连线最简单，这样的电路称为“最小化”电路。其次，为了满足速度要求，应使级数尽量少，以减少门电路的延时；电路的功耗应尽可能地小，工作时稳定可靠。

组合电路的描述方式有四种：真值表、逻辑代数、结构描述、抽象描述。采用 Verilog HDL 进行组合电路设计主要采用的就是这几种方式。下面结合具体的实例简单介绍这四种描述方式。

例 4.2-1 设计一个 3 个裁判的表决电路，当两个或两个以上裁判同意时，判决器输出“1”，否则输出“0”。

方法 1：真值表方式。

真值表是对电路功能最直接和简单的描述方式。根据电路的功能，可以通过真值表直接建立起输出与输入之间的逻辑关系。例 4.2-1 有三个输入端 A、B、C 和一个输出端 OUT，其真值表如表 4.2-1 所示。

表 4.2-1 例 4.2-1 真值表

A	B	C	OUT
0	0	0	0
0	0	1	0
0	1	0	0
1	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1

在 Verilog HDL 中，可以使用 case 语句对电路进行描述性设计，表 4.2-1 的真值表设计代码如下：

```
module design1(OUT, A, B, C);
    output OUT;
    input A, B, C;
```

```
  reg OUT;
    always @(A or B or C)
      case ({A, B, C})
        3'b000 : OUT <= 0;
        3'b001 : OUT <= 0;
        3'b010 : OUT <= 0;
            3'b100 : OUT <= 0;
        3'b011 : OUT <= 1;
        3'b101 : OUT <= 1;
        3'b110 : OUT <= 1;
        3'b111 : OUT <= 1;
      endcase
endmodule
```

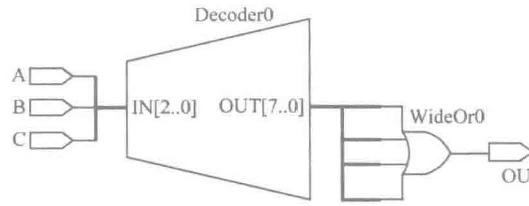


图 4.2-1　真值表设计方式的电路结构

真值表描述方式本质上是最小项表达式，该描述方式设计的电路结构如图 4.2-1 所示。

方法 2：逻辑代数方式。

组合电路的另一种表达方法是逻辑代数方式，其主要思想是将真值表用卡诺图来表示，然后化简电路，得出逻辑函数表达式。图 4.2-2 是例 4.2-1 的卡诺图。

通过对卡诺图的化简，可以得到组合电路逻辑输出与输入之间的逻辑函数表达式：

$$OUT = AB + BC + AC \tag{4.2-1}$$

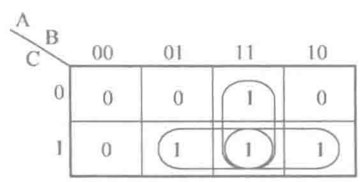


图 4.2-2　例 4.2-1 的卡诺图

根据逻辑函数表达式可以很方便地写出采用逻辑代数方式的 Verilog HDL 程序代码：

```
module design2(OUT, A, B, C);
  output OUT;
  input A, B, C;
    assign OUT = (A&B)|(B&C)|(A&C);
endmodule
```

方法 3：结构描述方式。

结构描述方式是对电路最直接的表示，早期的数字电路设计通常采用的原理图设计实际上就是一种结构描述方式。Verilog HDL 同样可以采用结构描述方式对数字电路进行设计。图 4.2-3 是公式(4.2-1)表示的基本电路单元构成的电路结构。

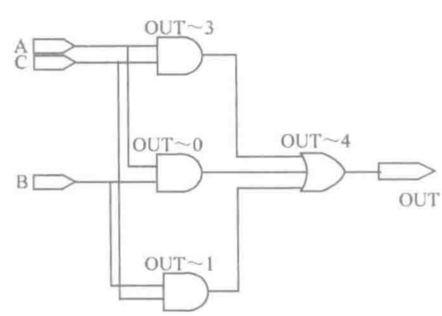


图 4.2-3　逻辑表达式 4.2-1 的电路结构

其 Verilog HDL 程序代码如下：

```
module design3(OUT, A, B, C);
  output OUT;
  input A, B, C;
      and U1 (w1, A, B);
      and U2 (w2, B, C);
```

```
        and U3 (w3, A, C);
        or U4 (OUT, w1, w2, w3);
endmodule
```

方法 4：抽象描述方式。

除了以上 3 种方法，Verilog HDL 还提供了以抽象描述方式进行电路设计的方法，可以直接从电路功能出发，编写代码。例如判决器设计，将三个输入的判决相加，当判决成功时相加器之和大于 1，即表示投票成功。采用这种描述方式的 Verilog HDL 程序代码如下：

```
module design4(OUT, A, B, C);
    output OUT;
    input A, B, C;
    wire [1:0] sum;
    reg OUT;
        assign sum = A+B+C;
            always @(sum)
                if (sum>1)        OUT = 1;
                else              OUT = 0;
endmodule
```

可以看到，以上 4 种 Verilog HDL 描述方式都可以对表决电路进行设计。这里应该指出的是，Verilog HDL 程序是对逻辑电路功能的描述性设计，并非最终得到的电路。EDA 综合工具可以将 Verilog HDL 程序综合成物理电路形式，通过电路优化，从而得到符合设计要求的最简化电路。采用 Synplify 软件对上面 4 种方式中任一种方式设计的 Verilog HDL 程序进行综合(采用 Altera 公司的 Stratix II 器件)，可以得到相同的最简化电路，如图 4.2-4 所示。

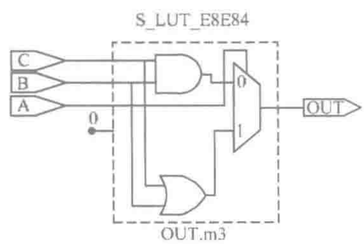


图 4.2-4　例 4.2-1 综合优化后的电路

Verilog HDL 对同一个电路有不同风格的描述，对于简单的电路，其各种描述之间的差异不大。但对于复杂的电路，不同风格的代码综合出的电路将会不同，生成的电路的性能也会不同。因此 Verilog HDL 是与电路对应的，要求设计者在编写代码时对所设计电路应有清楚的认识，这需要平时大量的积累。下面以典型的组合电路为例，分别对 Verilog HDL 基本数字逻辑组合电路的设计方法进行说明。

4.2.1　数字加法器

数字加法器是一种较为常用的逻辑运算器件，被广泛用于计算机、通信和多媒体数字集成电路中。广义的加法器包括加法器和减法器，在实际系统中加法器的输入通常采用的是补码形式，因此就电路结构而言，加法电路和减法电路是一样的，只不过输入信号采用的是补码输入。

例 4.2-2　2 输入 1 bit 信号全加器。

如果运算考虑了来自低位的进位，那么该运算就为全加运算。实现全加运算的电路称为全加器。2 输入 1 bit 信号全加器的真值表如表 4.2-2 所示。

表 4.2-2　2 输入 1 bit 信号全加器真值表

A	B	C_IN	SUM	C_OUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

代数逻辑表示为

$$SUM = A \oplus B \oplus C_IN \quad (4.2\text{-}2)$$

$$C_OUT = AB + (A \oplus B)C_IN \quad (4.2\text{-}3)$$

对应的电路如图 4.2-5 所示。

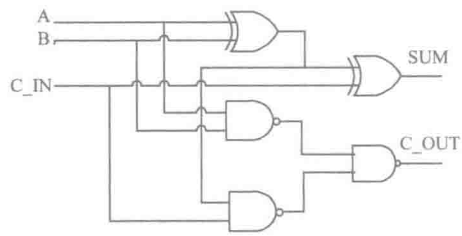


图 4.2-5　2 输入 1 bit 全加器电路

Verilog HDL 可以用不同的描述方式写出 1 bit 全加器，其综合电路是相同的，仅仅是描述风格不同。在此给出两种不同的风格：利用连续赋值语句实现和利用行为描述方式实现。

(1) 利用连续赋值语句实现。

```
module one_bit_fulladder(SUM, C_OUT, A, B, C_IN);
    input     A, B, C_IN;
    output    SUM, C_OUT;
        assign     SUM = (A^B)^C_IN;
        assign     C_OUT = (A&B)|((A^B)&C_IN);                //全加器的输出
endmodule
```

(2) 利用行为描述方式实现。

```
module one_bit_fulladder(SUM, C_OUT, A, B, C_IN);
    output    SUM, C_OUT;
    input     A, B, C_IN;
        assign      {C_OUT, SUM} = A+B+C_IN;
endmodule
```

采用行为描述方式可以提高设计的效率，对于一个典型的多位加法器的行为描述设计，仅需改变代码中输入和输出信号的位宽即可。例如一个 2 输入 8 bit 加法器，可以采用下面的 Verilog HDL 程序代码实现。

```
module eight_bits_fulladder(SUM, C_OUT, A, B, C_IN);
    output[7:0]   SUM;
    output        C_OUT;
    input [7:0]   A, B;
```

```
    input          C_IN;
        assign {C_OUT, SUM} = A+B+C_IN;
endmodule
```

例 4.2-3 4 位超前进位加法器示例。

超前进位加法器是一种高速加法器，每级进位由附加的组合电路产生，高位的运算不需等待低位运算完成，因此可以提高运算速度。

对于输入信号位宽为 N 的全加器，其进位信号是

$$C_OUT = C_N \tag{4.2-4}$$

输出的加法结果是

$$SUM_OUTN_{n-1} = P_{n-1} \oplus C_{n-1} \quad n \in [N,\ 1] \tag{4.2-5}$$

超前进位标志信号是

$$\left.\begin{aligned} C_n = G_{n-1} + P_{n-1}C_{n-1} \quad n \in [N, 1] \\ C_0 = C_IN \end{aligned}\right\} \tag{4.2-6}$$

进位产生函数是

$$G_{n-1} = A_{n-1}B_{n-1} \quad n \in [N,\ 1] \tag{4.2-7}$$

进位传输函数是

$$P_{n-1} = A_{n-1} \oplus B_{n-1} \quad n \in [1,\ N] \tag{4.2-8}$$

上述公式中 N 为加法器位数，在 4 位加法器中，N = 4。由此可以推出各级进位信号表达式，并构成快速进位逻辑电路。

$$\left.\begin{aligned} C_1 &= G_0 + P_0C_0 \\ C_2 &= G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}\right\} \tag{4.2-9}$$

4 位超前进位加法器的电路如图 4.2-6 所示。

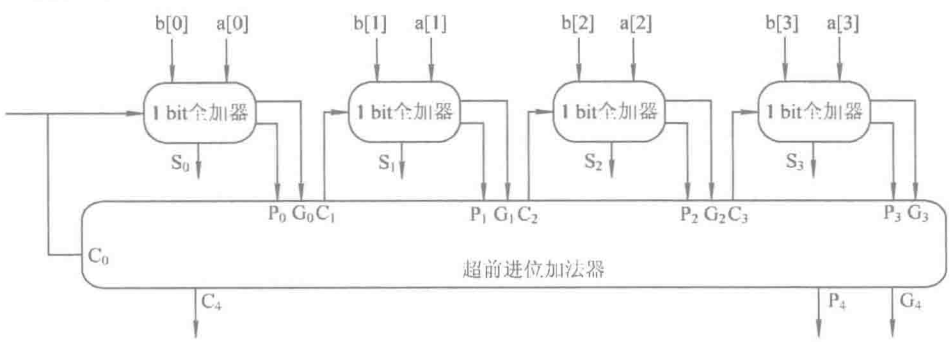


图 4.2-6 4 位超前进位加法器的电路

4 位超前进位加法器对应的 Verilog HDL 程序代码如下：

```
module four_bits_fast_addder (sum_out, c_out, a, b, c_in);
    input [3:0]   a, b;         //加数，被加数
    input         c_in;         //来自前级的进位
    output [3:0]  sum_out;      //和
```

```
    output        c_out;          //进位输出
    wire [4:0]    g, p, c;        //产生函数、传输函数和内部进位
    assign c[0] = c_in;
    assign p = a | b;
    assign g = a&b;
    assign c[1] = g[0] | (p[0]&c[0]);
    assign c[2] = g[1] | (p[1]&(g[0] | (p[0]&c[0])));
    assign c[3] = g[2] | (p[2]&(g[1] | (p[1]&(g[0] | (p[0]&c[0])))));
    assign c[4] = g[3] | (p[3]&(g[2] | (p[2]&(g[1] | (p[1]&(g[0]|(p[0]&c[0])))))));
    assign sum_out = p^c[3:0];
    assign c_out = c[4];
endmodule
```

4.2.2　数据比较器

数据比较器用来对两个二进制数的大小进行比较，或检测逻辑电路是否相等。数据比较器包含两部分功能：一是比较两个数的大小；二是检测两个数是否一致。

例 4.2-4　4 位数据比较器示例。

多位数据比较器的比较过程由高位到低位逐位进行，而且只有在高位相等时，才进行低位比较。4 位数据比较器中进行 $A_3A_2A_1A_0$ 和 $B_3B_2B_1B_0$ 的比较时，首先比较最高位 A_3 和 B_3。如果 $A_3 > B_3$，那么不管其它几位数为何值，结果均为 $A > B$；若 $A_3 < B_3$，结果为 $A < B$。如果 $A_3 = B_3$，就必须通过比较低一位 A_2 和 B_2 来判断 A 和 B 的大小。如果 $A_2 = B_2$，还必须通过比较更低一位 A_1 和 B_1 来判断大小，直到最后一位的比较。如果完全相等，则由前一级结果 C 确定。

为了便于扩展，比较器一般带有扩展端口。在输入端会有 $C_{A>B}$、$C_{A=B}$、$C_{A<B}$ 三个端口，将低一级的输出连接到对应的输入端，当高级的数据完全相等时，输出则依据 $C_{A>B}$、$C_{A=B}$、$C_{A<B}$ 这三个扩展端口决定。4 位数据比较器的真值表如表 4.2-3 所示。

表 4.2-3　4 位数据比较器的真值表

输　入							输　出		
A_3 B_3	A_2 B_2	A_1 B_1	A_0 B_0	$C_{A>B}$	$C_{A=B}$	$C_{A<B}$	$F_{A>B}$	$F_{A=B}$	$F_{A<B}$
$A_3>B_3$	x x	x x	x x	x	x	x	1	0	0
$A_3<B_3$	x x	x x	x x	x	x	x	0	0	1
$A_3=B_3$	$A_2>B_2$	x x	x x	x	x	x	1	0	0
$A_3=B_3$	$A_2<B_2$	x x	x x	x	x	x	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1>B_1$	x x	x	x	x	1	0	0
$A_3=B_3$	$A_2=B_2$	$A_1<B_1$	x x	x	x	x	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0>B_0$	x	x	x	1	0	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0<B_0$	x	x	x	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	$C_{A>B}$	$C_{A=B}$	$C_{A<B}$	$C_{A>B}$	$C_{A=B}$	$C_{A<B}$

用 F[2:0] 表示比较结果{$F_{A>B}$，$F_{A=B}$，$F_{A<B}$}，C[2:0] 表示前一级比较结果{$C_{A>B}$，$C_{A=B}$，

$C_{A<B}$}，采用描述方式的 Verilog HDL 程序代码如下：

```
module four_bits_comp1(F, A, B, C);
    parameter comp_width = 4;
    output [2:0] F;
    input [2:0] C;
    input [comp_width-1:0] A;
    input [comp_width-1:0] B;
    reg [2:0] F;
        always @(A or B or C)
            if (A>B)            F = 3'b100;
            else if (A<B)       F = 3'b001;
            else                F = C;
endmodule
```

4.2.3 数据选择器

数据选择器又称多路选择器(Multiplexer，简称 MUX)，它有 n 位地址输入、2^n 位数据输入、1 位数据输出。每次在输入地址的控制下，从多路输入数据中选择一路输出，其功能类似于一个单刀多掷开关，见图 4.2-7。在 ASIC 和 FPGA 的器件库中，通常都有不同输入端口的多路选择器，可以利用这些选择器构成功能更加复杂的数据选择电路。

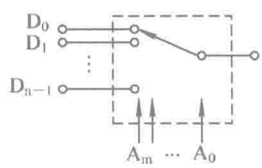


图 4.2-7　数据选择器框图及等效开关

例 4.2-5　8 选 1 数据选择器示例。

8 选 1 数据选择器可以由多个 2 选 1 数据选择器构成，也可以采用抽象描述方式进行设计；可以采用 2 选 1 数据选择器串行连接，也可以用树形连接分成三级实现。

(1) 多个 2 选 1 数据选择器的结构级描述(见图 4.2-8)。

```
module mux8to1_2(d_out, d_in, sel);
    output    d_out;
    input [7:0]    d_in;
    input [2:0]    sel;
    wire[3:0]    w1;
    wire[1:0]    w2;
      assign w1 = sel[0]? {d_in[7], d_in[5], d_in[3], d_in[1]} :{d_in[6], d_in[4], d_in[2], d_in[0]};
        assign w2 = sel[1]? {w1[3], w1[1]} :{w1[2], w1[0]};
        assign d_out = sel[2]?w2[1]:w2[0];
endmodule
```

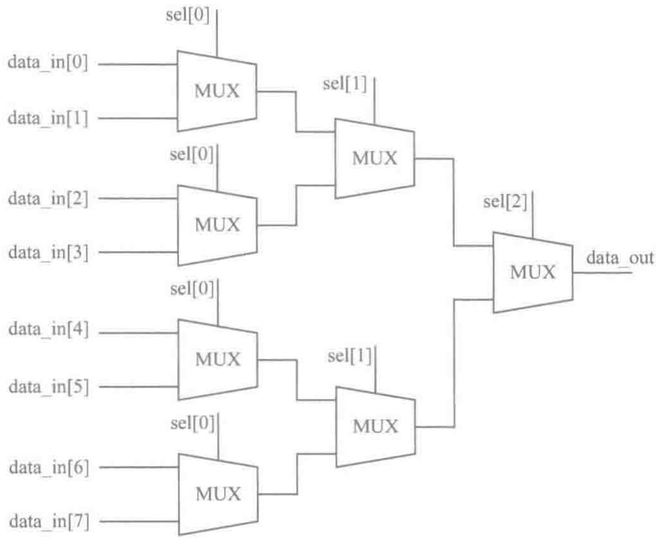


图 4.2-8　树形连接的 8 选 1 数据选择器电路

(2) 抽象描述方式。多路选择器的设计可以采用 case 语句直接进行设计。在这种设计方式中，只需考虑选择信号列表就可以实现功能更为复杂的数据选择器。

```
module mux8to1(out, sel, data_in);
    output out;
    input [7:0] data_in;
    input [3:0] sel;
    reg out;
        always @ (data_in or sel)
          case (sel)
            3'b000 : out <= data_in[0];
            3'b001 : out <= data_in[1];
            3'b010 : out <= data_in[2];
            3'b011 : out <= data_in[3];
            3'b100 : out <= data_in[4];
            3'b101 : out <= data_in[5];
            3'b110 : out <= data_in[6];
            3'b111 : out <= data_in[7];
          endcase
endmodule
```

4.2.4　数字编码器

用文字、符号或数码表示特定对象的过程称为编码。在数字电路中用二进制代码表示有关的信号称为二进制编码。实现编码操作的电路叫做编码器。

例 4.2-6 3 位二进制 8 线-3 线编码器示例。

用 n 位二进制代码对 $N = 2^n$ 个一般信号进行编码的电路，叫做二进制编码器。例如 n = 3，可以对 8 个一般信号进行编码。这种编码器的特点是：任何时刻只允许输入一个有效信号，不允许同时出现两个或两个以上的有效信号。假设编码器规定高电平为有效电平，则在任何时刻只有一个输入端为高电平，其余输入端为低电平。同理，如果规定低电平为有效电平，则在任何时刻只有一个输入端为低电平，其余输入端为高电平。因而其输入是一组有约束(互相排斥)的变量。

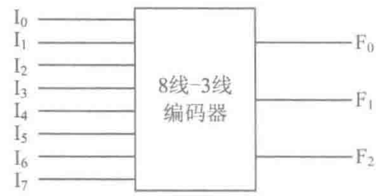


图 4.2-9 3 位二进制 8 线-3 线编码器框图

图 4.2-9 是 3 位二进制 8 线-3 线编码器框图，它的输入是 I_0～I_7，8 个高电平有效信号，输出是 3 位二进制代码 F_2、F_1、F_0。输出与输入的对应关系如表 4.2-4 所示。

表 4.2-4 3 位二进制 8 线-3 线编码器真值表

输入								输出		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	F_2	F_1	F_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

采用抽象描述方式的 Verilog HDL 程序代码如下：

```
module code_8to3(F, I);
    output [2:0] F;
    input [7:0] I;
    reg [2:0] F;
        always @ (I)
         case (I)
            8'b00000001: F = 3'b000;
            8'b00000010: F = 3'b001;
            8'b00000100: F = 3'b010;
            8'b00001000: F = 3'b011;
            8'b00010000: F = 3'b100;
            8'b00100000: F = 3'b101;
            8'b01000000: F = 3'b110;
            8'b10000000: F = 3'b111;
```

```
        default : F = 3'bx;
    endcase
endmodule
```

例 4.2-7 8 线-3 线优先编码器示例。

二进制编码器要求任何时刻只有一个输入有效，若同时有两个或更多个输入信号有效，将造成输出混乱，因此二进进编码器在使用过程中有一定局限性。克服这种局限性的一种方法是采用优先编码器。优先编码器允许多个输入信号同时有效，但它只对其中优先级别最高的有效输入信号编码，对级别低的输入信号不予理睬。优先编码器常用于优先中断系统和键盘编码。8 线-3 线优先编码器的逻辑符号如图 4.2-10 所示，它有 8 个输入端 $\bar{I}_0 \sim \bar{I}_7$，低电平为输入有效电平；3 个输出端 $\overline{Y}_0 \sim \overline{Y}_2$，低电平为输出有效电平。此外，为了便于电路的扩展和使用的灵活，还设置有使能端 $\overline{S}$、选通输出端 $\overline{Y}_s$ 和扩展端 $\overline{Y}_{ex}$。8 线-3 线优先编码器的真值表见表 4.2-5。

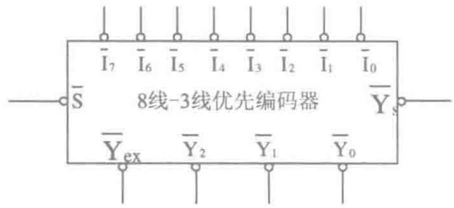


图 4.2-10 8 线-3 线优先编码器的逻辑符号

表 4.2-5 8 线-3 线优先编码器的真值表

输入									输出				
$\overline{S}$	$\bar{I}_0$	$\bar{I}_1$	$\bar{I}_2$	$\bar{I}_3$	$\bar{I}_4$	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$	$\overline{Y}_s$	$\overline{Y}_{ex}$
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	1	0
0	x	x	x	x	x	x	0	1	0	0	1	1	0
0	x	x	x	x	x	0	1	1	0	1	0	1	0
0	x	x	x	x	0	1	1	1	0	1	1	1	0
0	x	x	x	0	1	1	1	1	1	0	0	1	0
0	x	x	0	1	1	1	1	1	1	0	1	1	0
0	x	0	1	1	1	1	1	1	1	1	0	1	0
0	0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0	1

功能表说明：$\overline{S}=1$ 时，电路处于禁止工作状态，此时无论 8 个输入为何种状态，3 个输出端均为高电平，$\overline{Y}_s$ 和 $\overline{Y}_{ex}$ 也为高电平，编码器不工作。当 $\overline{S}=0$ 时，电路处于正常工作状态，允许 $\bar{I}_0 \sim \bar{I}_7$ 当中同时有几个输入端为低电平，即同时有几路编码输入信号有效。在 8 个输入端中，$\bar{I}_7$ 的优先权最高，$\bar{I}_0$ 的优先权最低。当 $\bar{I}_7=0$ 时，无论其它输入端有否有效输入信号(功能表中以 x 表示)，输出端只输出 $\bar{I}_7$ 的编码，即 $\overline{Y}_2\overline{Y}_1\overline{Y}_0=000$；当 $\bar{I}_7=1$，$\bar{I}_6=0$ 时，无论其余输入端有否有效输入信号，只对 $\bar{I}_6$ 进行编码，输出为 $\overline{Y}_2\overline{Y}_1\overline{Y}_0=001$，其余状态依次类推。表中出现的 3 种输出 $\overline{Y}_2\overline{Y}_1\overline{Y}_0=111$ 的情况可以用 $\overline{Y}_s$ 和 $\overline{Y}_{ex}$ 的不同状态来区别，即如果 $\overline{Y}_2\overline{Y}_1\overline{Y}_0=111$ 且 $\overline{Y}_s\ \overline{Y}_{ex}=10$，则表示电路处于工作状态而且 $\bar{I}_0$ 有编码信号输入；如

果 $\overline{Y}_2\overline{Y}_1\overline{Y}_0 = 111$ 且 $\overline{Y}_s\ \overline{Y}_{ex} = 01$，则表示电路处于工作状态但没有输入编码信号。由于没有输入编码信号时 $\overline{Y}_s = 0$，因此 $\overline{Y}_s$ 也可以称为“无编码输入信号”。

8 线-3 线优先编码器的 Verilog HDL 程序代码如下：

```
module mux8to3_p(data_out, Ys, Yex, sel, data_in);
    output [2:0] data_out;
    output Ys, Yex;
    input [7:0] data_in;
    input sel;
    reg [2:0] data_out;
    reg Ys, Yex;
        always @ (data_in or sel)
          if (sel) {data_out, Ys, Yex} = {3'b111, 1'b1, 1'b1};
          else
            begin
              casex (data_in)
                8'b0??????? : {data_out, Ys, Yex} = {3'b000, 1'b1, 1'b0};
                8'b10?????? : {data_out, Ys, Yex} = {3'b001, 1'b1, 1'b0};
                8'b110????? : {data_out, Ys, Yex} = {3'b010, 1'b1, 1'b0};
                8'b1110???? : {data_out, Ys, Yex} = {3'b011, 1'b1, 1'b0};
                8'b11110??? : {data_out, Ys, Yex} = {3'b100, 1'b1, 1'b0};
                8'b111110?? : {data_out, Ys, Yex} = {3'b101, 1'b1, 1'b0};
                8'b1111110? : {data_out, Ys, Yex} = {3'b110, 1'b1, 1'b0};
                8'b11111110 : {data_out, Ys, Yex} = {3'b111, 1'b1, 1'b0};
                8'b11111111 : {data_out, Ys, Yex} = {3'b111, 1'b0, 1'b1};
              endcase
            end
endmodule
```

例 4.2-8　二进制转化十进制 8421BCD 编码器示例。

将十进制数 0、1、2、3、4、5、6、7、8、9 这 10 个信号编成二进制代码的电路叫做二进制转化十进制编码器。其输入代表的是 0～9 这 10 个数字的状态信息，有效信号为 1(即某信号为 1 时，则表示要对它进行编码)，输出是相应的 BCD 码，因此也称 10 线-4 线编码器。它和二进制编码器一样，任何时刻只允许输入一个有效信号。

式(4.2-10)为 8421BCD 编码器的逻辑表达式，对应的真值表见表 4.2-6。

$$\begin{cases} D = Y_8 + Y_9 = \overline{\overline{Y}_8 \cdot \overline{Y}_9} \\ C = Y_4 + Y_5 + Y_6 + Y_7 = \overline{\overline{Y}_4 \cdot \overline{Y}_5 \cdot \overline{Y}_6 \cdot \overline{Y}_7} \\ B = Y_2 + Y_3 + Y_6 + Y_7 = \overline{\overline{Y}_2 \cdot \overline{Y}_3 \cdot \overline{Y}_6 \cdot \overline{Y}_7} \\ A = Y_1 + Y_3 + Y_5 + Y_7 + Y_9 = \overline{\overline{Y}_1 \cdot \overline{Y}_3 \cdot \overline{Y}_5 \cdot \overline{Y}_7 \cdot \overline{Y}_9} \end{cases} \tag{4.2-10}$$

表 4.2-6　8421BCD 编码器的真值表

十进制数	D	C	B	A
0(Y_0)	0	0	0	0
1(Y_1)	0	0	0	1
2(Y_2)	0	0	1	0
3(Y_3)	0	0	1	1
4(Y_4)	0	1	0	0
5(Y_5)	0	1	0	1
6(Y_6)	0	1	1	0
7(Y_7)	0	1	1	1
8(Y_8)	1	0	0	0
9(Y_9)	1	0	0	1

可以看到，对于这样一个规模的电路，可以用逻辑代数方式和结构描述方式进行电路设计，但是效率不高。通过对功能的抽象描述，在综合工具的帮助下，可以得到期望的电路。其 Verilog HDL 程序代码如下：

```
module BCD8421(data_out, data_in);
    output [3:0] data_out;
    input [8:0] data_in;
    reg [3:0] data_out;
        always @ (data_in)
          case (data_in)
            9'b000000000 : data_out = 4'b0000;
            9'b000000001 : data_out = 4'b0001;
            9'b000000010 : data_out = 4'b0010;
            9'b000000100 : data_out = 4'b0011;
            9'b000001000 : data_out = 4'b0100;
            9'b000010000 : data_out = 4'b0101;
            9'b000100000 : data_out = 4'b0110;
            9'b001000000 : data_out = 4'b0111;
            9'b010000000 : data_out = 4'b1000;
            9'b100000000 : data_out = 4'b1001;
            default      : data_out = 4'b0000;
      endcase
endmodule
```

例 4.2-9　8421BCD 十进制余 3 码编码器示例。

和 8421BCD 编码一样，余 3 码也是一种 BCD 编码，这种编码的特点是：用余 3 码作十进制加法运算时，若两数之和是 10，正好等于二进制数的 16，则从高位自动产生进位信

号。因此可以使用余 3 码简化计算。在 8421BCD 码上加 3 就得到了余 3 码。表 4.2-7 是余 3 码的真值表与 8421BCD 码的对比，图 4.2-11 是 8421BCD 转余 3 码编码器的逻辑符号及功能。

表 4.2-7 余 3 码与 8421BCD 码的真值表对比

十进制数	8421BCD 码	余 3 码	十进制数	8421BCD 码	余 3 码
0	0000	0011	5	0101	1000
1	0001	0100	6	0110	1001
2	0010	0101	7	0111	1010
3	0011	0110	8	1000	1011
4	0100	0111	9	1001	1100



图 4.2-11 8421BCD 转余 3 码编码器的逻辑符号及功能

和 8421BCD 编码器一样，余 3 码编码器也可以通过查找表的方式进行描述，仅需改变表 4.2-6 中真值表的内容即可。另外，也可以通过 8421BCD 码加“3”的方式得到，Verilog HDL 程序代码如下：

```
module code_change(B_out, B_in);
    output [3:0] B_out;
    input [3:0] B_in;
      assign B_out = B_in+2'b11;
endmodule
```

4.2.5 数字译码器

译码是编码的逆过程，它将二进制代码所表示的信息翻译成相应的状态信息。实现译码功能的电路称为译码器。这里只介绍二进制译码器。

N 位二进制译码器有 N 个输入端和 2^N 个输出端，一般称为 N 线-2^N 线译码器。常见的译码器有 2 线-4 线译码器、3 线-8 线译码器和 4 线-16 线译码器。

图 4.2-12 为 2 线-4 线译码器的逻辑电路及逻辑符号，其真值表如表 4.2-8 所示。图中，A_1、A_0 为地址输入端，A_1 为高位。$\overline{Y_0}$、$\overline{Y_1}$、$\overline{Y_2}$、$\overline{Y_3}$ 为状态信号输出端，非号表示低电平有效。E 为使能端(或称选通控制端)，低电平有效。当 E = 0 时，允许译码器工作，$\overline{Y_0}$～$\overline{Y_3}$ 中只允许一个为有效电平输出；当 E = 1 时，禁止译码器工作，所有输出 $\overline{Y_0}$～$\overline{Y_3}$ 均为高电平。一般使能端有两个用途：一是引入选通脉冲，以抑制冒险脉冲的发生；

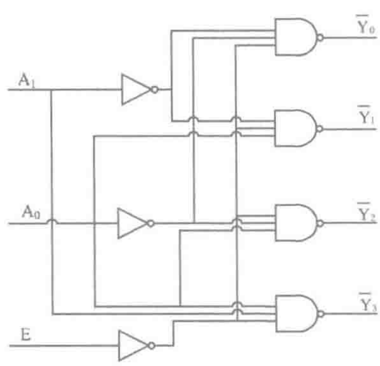


图 4.2-12 2 线-4 线译码器的逻辑电路及逻辑符号

二是用来扩展输入的变量数(功能扩展)。

表 4.2-8 2 线-4 线译码器的真值表

E	A_1	A_0	$\overline{Y}_0$	$\overline{Y}_1$	$\overline{Y}_2$	$\overline{Y}_3$
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

从表中可以看出，当 E = 0 时，2 线-4 线译码器的输出函数分别为 $\overline{Y}_0=\overline{\overline{A}_1\overline{A}_0}$，$\overline{Y}_1=\overline{\overline{A}_1 A_0}$，$\overline{Y}_2=\overline{A_1\overline{A}_0}$，$\overline{Y}_3=\overline{A_1 A_0}$，如果用 $\overline{Y}_i$ 表示 i 端的输出，m_i 表示输入地址变量 A_1、A_0 的一个最小项，则输出函数可写成 $\overline{Y}_i=\overline{\overline{E}m_i}$ (i = 0，1，2，3)。可见，译码器的每一个输出函数对应输入变量的一组取值，当使能端有效(E = 0)时，它正好是输入变量最小项的非。因此变量译码器也称最小项发生器。

同编码器一样，译码器的级联扩展也可使用同样的方法，其 Verilog HDL 程序代码如下：

```
module decode_2to4(Y, E, A);
    output [3:0] Y;
    input [1:0] A;
    input E;
    assign Y[0] = ~(~E&~A[1]&~A[0]);
    assign Y[1] = ~(~E&~A[1]&A[0]);
    assign Y[2] = ~(~E&A[1]&~A[0]);
    assign Y[3] = ~(~E&A[1]&A[0]);
endmodule
```

也可以采用抽象描述方式进行设计，其 Verilog HDL 程序代码如下：

```
module decode_2to4(Y, E, A);
    output [3:0] Y;
    input [1:0] A;
    input E;
    reg [3:0] Y;
        always @(E or A)
            case ({E, A})
                3'b1?? : Y = 4'b0000;
                3'b000 : Y = 4'b0001;
                3'b001 : Y = 4'b0010;
                3'b010 : Y = 4'b0100;
                3'b011 : Y = 4'b1000;
                default : Y = 4'b0000;
```

```
        endcase
    endmodule
```

采用抽象描述方式的优势是不需要对电路化简，根据相对固定的设计模式，可以直接得到所需要的电路。

4.2.6 奇偶校验器

奇偶校验器的功能是检测数据中所含“1”的个数是奇数还是偶数。在计算机和一些数字通信系统中，常用奇偶校验器来检查数据传输和数码记录中是否存在错误。

奇偶校验包含两种方式：奇校验和偶校验。奇校验保证传输数据和校验位中“1”的总数为奇数。如果数据中包含奇数个“1”，则校验位置“0”；如果数据中包含偶数个“1”，则校验位置“1”。例如，需要传输 1100111，数据中包含 5 个“1”，采用奇校验，校验位为“0”，将“11001110”传输给接收机。

偶校验保证传输数据和校验位中“1”的总数为偶数。如果数据中包含奇数个“1”，则校验位置“1”，如果数据中包含偶数个“1”，则校验位置“0”。例如：要传输 1100111，数据中包含 5 个“1”，采用偶校验，校验位为“1”，将“11001111”传输给接收机。

奇偶校验只能检测部分传输错误，它不能确定错误发生在哪位或哪几位，所以不能进行错误校正。当数据发生错误时只能重新发送数据。

奇偶校验一般用在能够重新操作的计算机硬件中，例如：SCSI 总线和微处理器中的高速缓存，在发生错误时，这些部件可以丢掉数据，获取重发数据。

例 4.2-10 8 bit 奇偶校验器示例。

8 bit 奇偶校验器的原理图如图 4.2-13 所示。

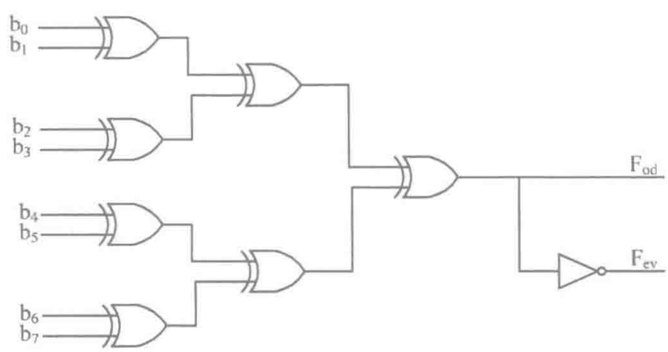


图 4.2-13 8 bit 奇偶校验器的原理图

图 4.2-13 中，校验器的输入 b_0～b_7 由 7 bit 数据和 1 bit 校验位组成。F_{od} 为判奇输出，F_{ev} 为判偶输出。当采用奇校验时，$F_{od}=1$，$F_{ev}=0$；当采用偶校验时，$F_{od}=0$，$F_{ev}=1$。

输出表达式为

$$\begin{cases} F_{od} = b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7 \\ F_{ev} = \overline{b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7} \end{cases} \tag{4.2-11}$$

例如：采用奇校验检测“1100111”，数据包含 5 个“1”，校验位为“0”，校验器的输入 b_0～b_7 为“11001110”，$F_{od}=1$，$F_{ev}=0$。

在 Verilog HDL 中，可以采用结构描述方式，也可以采用抽象描述方式。

(1) 结构描述方式：

```
module checker (Fod, Fev, b);
    output Fod, Fev;
    input [7:0] b;
        wire w1, w2, w3, w4, w5, w6;
            xor U1 (w1, b[0], b[1]);
            xor U2 (w2, b[2], b[3]);
            xor U3 (w3, b[4], b[5]);
            xor U4 (w4, b[6], b[7]);
            xor U5 (w5, w1, w2);
            xor U6 (w6, w3, w4);
            xor U7 (Fod, w5, w6);
            not U8 (Fev, Fod);
endmodule
```

(2) 抽象描述方式：

```
module checker (Fod, Fev, b);
    output Fod, Fev;
    input [7:0] b;
    assign Fod = ^b;
    assign Fev = ~Fod;
endmodule
```

4.3 时序电路的设计

与组合逻辑电路不同，时序逻辑电路的输出不仅与当前时刻输入变量的取值有关，而且与电路的原状态(即过去的输入情况有关)。

图 4.3-1 是时序逻辑电路的结构框图。与组合逻辑电路相比，时序逻辑电路有两个特点：第一，时序逻辑电路包括组合逻辑电路和存储电路两部分，存储电路具有记忆功能，通常由触发器组成；第二，存储电路的状态反馈到组合逻辑电路输入端，与外部输入信号共同决定组合逻辑电路的输出。组合逻辑电路的输出除包括外部输出外，还包含连接到存储电路的内部输出，它将控制存储电路状态的转移。



图 4.3-1　时序逻辑电路的结构框图

图 4.3-1 中，$X(x_1, x_2, \cdots, x_n)$是外部输入信号；$Q(q_1, q_2, \cdots, q_j)$是存储电路的状态输出，也是组合逻辑电路的内部输入；$Z(z_1, z_2, \cdots, z_m)$为外部输出信号；$Y(y_1, y_2, \cdots, y_k)$为存储电路的激励信号，也是组合逻辑电路的内部输出。在存储电路中，每一位输出 $q_i(i = 1, 2, \cdots, j)$称为一个状态变量，j 个状态变量可以组成 2^j 个不同的内部状态。时序逻辑电路对于输入变量历史情况的记忆反映在状态变量的不同取值上，即不同的内部状态代表不同的输入变量的历史情况。

以上四组信号之间的逻辑关系可用以下三个方程组来描述：

$$\begin{cases} z_1^n = f_1(x_1^n, x_2^n, \ldots, x_n^n, q_1^n, q_2^n, \ldots, q_j^n) \\ z_2^n = f_2(x_1^n, x_2^n, \ldots, x_n^n, q_1^n, q_2^n, \ldots, q_j^n) \\ \qquad \vdots \\ z_m^n = f_m(x_1^n, x_2^n, \ldots, x_n^n, q_1^n, q_2^n, \ldots, q_j^n) \end{cases} \tag{4.3-1}$$

$$\begin{cases} y_1^n = g_1(x_1^n, x_2^n, \ldots, x_n^n, q_1^n, q_2^n, \ldots, q_j^n) \\ y_2^n = g_2(x_1^n, x_2^n, \ldots, x_n^n, q_1^n, q_2^n, \ldots, q_j^n) \\ \qquad \vdots \\ y_k^n = g_k(x_1^n, x_2^n, \ldots, x_n^n, q_1^n, q_2^n, \ldots, q_j^n) \end{cases} \tag{4.3-2}$$

$$\begin{cases} q_1^{n+1} = h_1(y_1^n, y_2^n, \ldots, y_n^n, q_1^n, q_2^n, \ldots, q_j^n) \\ q_2^{n+1} = h_2(y_1^n, y_2^n, \ldots, y_n^n, q_1^n, q_2^n, \ldots, q_j^n) \\ \qquad \vdots \\ q_j^{n+1} = h_j(y_1^n, y_2^n, \ldots, y_n^n, q_1^n, q_2^n, \ldots, q_j^n) \end{cases} \tag{4.3-3}$$

式(4.3-1)是输出方程，式(4.3-2)是驱动方程(或称为激励方程)，式(4.3-3)是状态方程。方程中的上标 n 和 n+1 表示相邻的两个离散时间(或称为相邻的两个节拍)，如 q_1^n，q_2^n，…，q_j^n 表示存储电路中每个触发器的当前状态(也称为现状态或原状态)，q_1^{n+1}，q_2^{n+1}，…，q_j^{n+1} 表示存储电路中每个触发器的新状态(也称为下一状态或次状态)。以上三个方程组可写成如下形式：

$$Z^n = F(X^n, Q^n) \tag{4.3-4}$$

$$Y^n = G(X^n, Q^n) \tag{4.3-5}$$

$$Q^{n+1} = H(Y^n, Q^n) \tag{4.3-6}$$

时序逻辑电路某时刻的输出 Z^n 决定于该时刻的外部输入 X^n 和 Q^n。时序逻辑电路的工作过程实质上就是不同输入条件下内部状态不断更新的过程。

时序逻辑电路按状态变化的特点，可分为同步时序逻辑电路和异步时序逻辑电路。在同步时序逻辑电路中，电路状态的变化在同一时钟脉冲作用下发生，即各触发器状态的转换同步完成。在异步时序逻辑电路中，没有统一的时钟脉冲信号，即各触发器状态的转换是异步完成的。目前大多数数字电路是同步时序逻辑电路，本书所举例程都集中在同步时序逻辑电路上。

传统的时序逻辑电路的设计就是根据逻辑设计命题的要求，选择适当的器件，设计出合理的逻辑电路。Verilog HDL 对于时序逻辑电路的描述性设计与传统的电路设计紧密相关。同步时序逻辑电路的一般设计过程可以按图 4.3-2 所示的步骤进行。

对时序逻辑电路功能的描述方式主要有三种：逻辑方程，状态转移表和状态转移图，时序图。

同组合逻辑电路相似，采用 Verilog HDL 对时序逻辑电路的设计进行描述也有不同的方式，归纳起来主要有三种：状态转移图描述，基于状态化简的结构性描述，Verilog HDL 抽象描述。

下面结合具体的实例简单介绍三种不同的描述方式。



图 4.3-2　同步时序逻辑电路设计流程

例 4.3-1　用 Verilog HDL 设计一个“111”序列检测器，当输入三个或三个以上的“1”时，电路输出为 1，否则为 0。

(1) 状态转移图描述方式。首先确定电路的输入变量和输出变量。该电路仅有一个输入端和一个输出端，设输入变量为 X，代表输入序列；输出变量为 Z，表示检测结果。可以得到的状态转移图如图 4.3-3 所示。

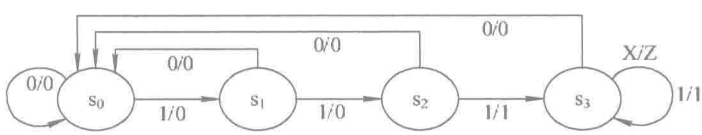


图 4.3-3　“111”序列检测器的状态转移图

因此设计时需要定义四种电路状态，设置如下：

s_0——初始状态，表示电路还未收到一个有效的 1。

s_1——电路收到了一个 1。

s_2——电路收到了连续的两个 1。

s_3——电路收到了连续的三个 1。

根据状态转移图，Verilog HDL 程序代码如下：

```
module checker(Z, X, clk);
    parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b11, s3 = 2'b10;
    output Z;
    input X, clk;
    reg [1:0] state, next_state;
    reg Z;
    always @(X, state)
      case (state)
        s0 : if (X)
              begin
```

```
            next_state <= s1;   Z = 0;
          end
        else
          begin
            next_state <= s0;   Z = 0;
          end
    s1 : if (X)
          begin
            next_state <= s2;   Z = 0;
          end
        else
          begin
            next_state <= s0;   Z = 0;
          end
    s2 : if (X)
          begin
            next_state <= s3;   Z = 1;
          end
        else
          begin
            next_state <= s0;   Z = 0;
          end
    s3 : if (X)
          begin
            next_state <= s3;   Z = 1;
          end
        else
          begin
            next_state <= s0;   Z = 0;
          end
  endcase
 always @(posedge clk)
   state <= next_state;
endmodule
```

用状态转移图描述电路时，不需要考虑电路的具体结构，只需了解电路的功能，掌握电路的工作流程即可，然后划分状态，分配寄存器，但是在状态划分后并没有对状态进行化简。因此，状态转移图不一定是最简单的电路形式。

用综合工具对上面的代码进行综合，可以得到如图 4.3-4 所示的电路。

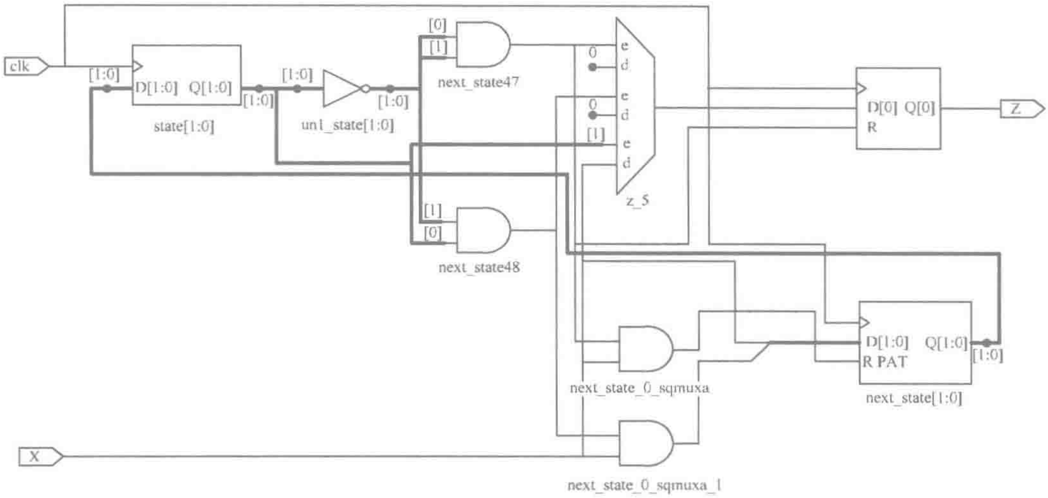


图 4.3-4　用综合工具综合后的电路

(2) 基于状态化简的结构性描述方式。对图 4.3-3 所示的状态转移图进行化简，仅剩三个状态，需要用两位二进制数表示，即需要两个 D 触发器储存状态。设 Q_1 表示高位寄存器的输出，Q_0 表示低位寄存器的输出。将状态的跳转以及输出 Z 用卡诺图的形式示出，如图 4.3-5 所示。

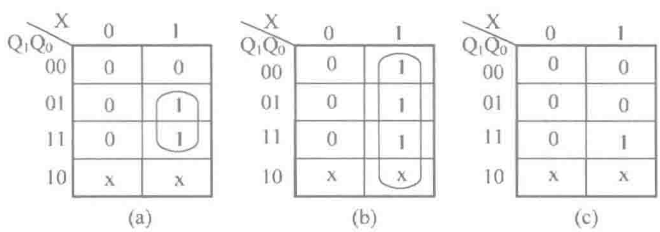


图 4.3-5　Q_1^{n+1} 和 Q_0^{n+1} 及输出 Z 的卡诺图

由卡诺图可以得出电路的输出方程和状态方程：

$$Q_1^{n+1} = Q_0 \tag{4.3-7}$$

$$Q_0^{n+1} = X \tag{4.3-8}$$

$$Z = Q_1Q_0X \tag{4.3-9}$$

根据化简的状态方程，Verilog HDL 程序代码如下：

```
//序列检测器模块
module checker (Z, X, clk);
    output Z;
    input X, clk;
    wire w1, w2;
        DFF U1 (.clk(clk), .D(X), .Q(w1));
        DFF U2 (.clk(clk), .D(w1), .Q(w2));
```

```
        assign Z = X & w1 & w2;
endmodule

//D 触发器模块
module DFF (Q, D, clk);
    output Q;
    input D, clk;
    reg Q;
        always @(posedge clk)   Q <= D;
endmodule
```

用综合工具对上面的代码进行综合，可以得到图 4.3-6 所示的电路。

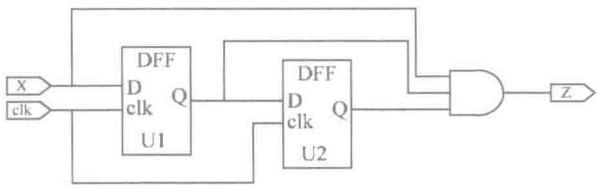


图 4.3-6　用综合工具综合后的电路

(3) Verilog HDL 抽象描述方式。在 Verilog HDL 中还可以对电路进行抽象描述。实现序列“111”的检测，可以使用 1 个三位的移位寄存器，将输入 X 作为移位寄存器的输入，当寄存器中为 111 时，输出 Z 为 1。Verilog HDL 程序代码如下：

```
module checker (Z, X, clk);
    output Z;
    input X, clk;
    reg [2:0] q;
    reg Z;
      always @(posedge clk)       q <= {q[1:0], X};
      always @(posedge clk)
            if (q==3'b111)        Z = 1;
                 else             Z = 0;
endmodule
```

用综合工具对上面的代码进行综合，可以得到图 4.3-7 所示的电路。

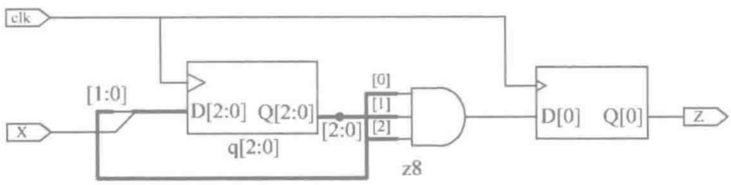


图 4.3-7　用综合工具综合后的电路

由上述三种描述方式和综合电路可知，不同电路描述方式所产生的综合电路不尽相同，

这也从另一个角度体现了数字电路设计的灵活性。下面根据典型的数字时序逻辑电路对 Verilog HDL 基本数字时序逻辑电路的设计方法分别进行说明。

4.3.1 触发器

触发器是时序逻辑电路的最基本电路单元，主要有 D 触发器、JK 触发器、T 触发器和 RS 触发器等。根据功能要求的不同，触发器还具有置位、复位、使能、选择等功能。

例 4.3-2 最简 D 触发器示例。

D 触发器的逻辑符号如图 4.3-8 所示，图中 D 为信号输入端，clk 为时钟控制端，Q 为信号输出端。这种触发器的逻辑功能是：不论触发器原来的状态如何，输入端的数据 D(无论 D = 0，还是 D = 1)都将在时钟 clk 的上升沿被送入触发器，使得 Q = D。其特征方程可描述为 $Q_{n+1} = D_n$。其真值表如表 4.3-1 所示。

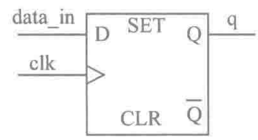


图 4.3-8 D 触发器的逻辑符号

表 4.3-1 D 触发器的真值表

clk	D	Q_n	Q_{n+1}
0	x	0	0
0	x	1	1
1	x	0	0
1	x	1	1
↑	0	0	0
↑	0	1	0
↑	1	0	1
↑	1	1	1

其 Verilog HDL 程序代码如下：

```
module DFF(q, clk, data_in);
    output q;
    input clk, data_in;
    reg q;
        always @(posedge clk)      q <= data_in;
endmodule
```

例 4.3-3 带复位端的 D 触发器示例。

在 D 触发器的实际使用中，有时需要一个复位端(也称清零端)。带有复位端的 D 触发器的逻辑符号如图 4.3-9 所示。

电路上电时，电路的逻辑处于不定状态，复位脉冲的到来将电路初始化为 Q = 0 的状态。随后，在时钟的控制下，输入端 D 的数据在每个时钟上升沿被置到输出端 Q。同步清 0 的 D 触发器的 Verilog HDL 程序代码如下：

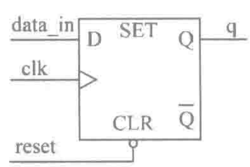


图 4.3-9 带复位端的 D 触发器的逻辑符号

```
module DFF_rst (q, clk, reset, data_in);
    output q;
    input clk, reset, data_in;
    reg q;
    always @(posedge clk)
        if (!reset)      q <= 0;
        else             q <= data_in;
endmodule
```

异步清 0 的 D 触发器的 Verilog HDL 程序代码如下：

```
module DFF_srst (q, clk, reset, data_in);
    output q;
    input clk, reset data_in;
    reg q;
        always @(posedge clk or negedge reset)
            if (!reset)  q <= 0;
            else         q <= data_in;
endmodule
```

可以看到，同步清 0 和异步清 0 触发器的电路代码只是在 always 后的敏感向量表上有所不同。Verilog HDL 是描述电路的，编写程序代码的时候一定要建立代码和电路对应的思想。对于同步清 0，并不是清 0 信号一变化电路马上就会被置 0，清 0 信号有效后需等待时钟的有效边沿到来后电路才会有动作，因此不应把清 0 信号写入敏感向量表中。而异步清 0 时，只要清 0 信号有效，电路就会马上更新，输出置 0，因此对于异步电路，清 0 信号有必要写入敏感向量表中。

例 4.3-4　复杂功能的 D 触发器示例。

前面介绍了最简单的 D 触发器和带有同步清 0、异步清 0 的 D 触发器。这里给出同步清 0 和异步清 0 共同在一个触发器上的复杂 D 触发器例子。其 Verilog HDL 程序代码如下：

```
module DFF_1(q, clk, reset1, reset2, data_in);
    output q;
    input clk, reset1, reset2, data_in;
    reg q;
        always @(posedge clk)
            if (!reset1)     q <= 0;
            else             q <= data_in;
        always @(posedge clk or negedge reset2)
            if (!reset2)     q <= 0;
            else             q <= data_in;
endmodule
```

例 4.3-5　T 触发器示例。

T 触发器的逻辑符号如图 4.3-10 所示，其逻辑功能为：当时钟的有效边沿到来时，如

果 T = 1，则触发器翻转；如果 T = 0，则触发器的状态保持不变。reset 为复位端，异步复位，低电平有效。

T 触发器的 Verilog HDL 程序代码如下：

```
module TFF(data_out, T, clk, reset);
    output data_out;
    input T, clk, reset;
    reg data_out;
        always @(posedge clk or negedge reset)
          if (!reset)      data_out <= 1'b0;
          else   if (T)         data_out <= ~data_out;
endmodule
```

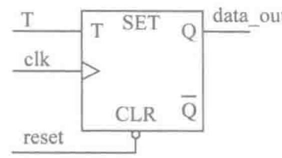


图 4.3-10　T 触发器的逻辑符号

4.3.2　计数器

计数器是应用最广泛的逻辑部件之一。计数器可以统计输入脉冲的个数，具有计时、计数、分频、定时、产生节拍脉冲等功能。

计数器的种类繁多，根据计数器中触发器时钟端的链接方式，分为同步计数器和异步计数器；根据计数方式，分为二进制计数器、十进制计数器和任意进制(也称为 M 进制)计数器；根据计数器中的状态变化规律，分为加法计数器、减法计数器和加/减计数器。

例 4.3-6　二进制计数器示例。

图 4.3-11 是采用 D 触发器设计二进制计数器的逻辑电路图。

由 D 触发器实现的二进制计数器的 Verilog HDL 程序代码如下：

```
module comp2bit (Q, clk, reset);
    output Q;
    input clk, reset;
    reg Q;
        always @ (posedge clk or negedge reset)
          if (!reset)
              Q <= 1'b0;
          else
              Q <= ~Q;
endmodule
```

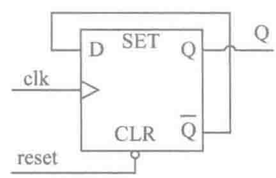


图 4.3-11　二进制计数器的逻辑电路图

例 4.3-7　任意进制计数器示例。

在数字电路系统中，经常会使用任意进制计数器，Verilog HDL 可以很好地支持不同进制计数器的设计。

对于 M 进制的计数器，首先应确定计数器所需触发器的个数。N 个触发器对应了 2^N 个状态，应有 2^N>M。任意进制计数器选取满足条件的最小 N，N 为计数器中触发器的个数。计数器的状态跳转有两种实现方法：反馈清零法和反馈置数法。

以十一进制计数器为例，最少需要 4 个触发器。采用反馈清零法设计的十一进制计数器的 Verilog HDL 程序代码如下：

```
module comp_11(count, clk, reset);
    output [3:0] count;
    input clk, reset;
    reg [3:0] count;
    always @ (posedge clk)
      if (reset)    count <= 4'b0000;
      else
        if (count == 4'b1010)
          count <= 4'b0000;
        else
          count <= count+1;
endmodule
```

4.3.3 移位寄存器

移位寄存器可以用来实现数据的串并转换，也可以构成移位行计数器，进行计数、分频，还可以构成序列码发生器、序列码检测器等，它也是数字系统中应用非常广泛的时序逻辑部件之一。

例 4.3-8 环形移位寄存器示例。

N 位环型寄存器由 N 个移位寄存器组成，它可以实现环形移位，如图 4.3-12 所示。

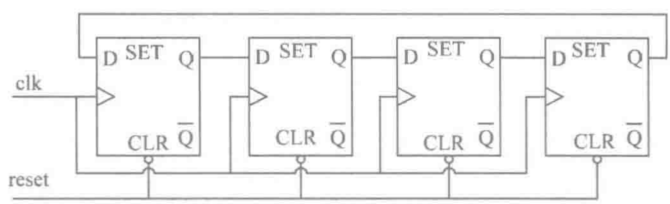


图 4.3-12　环形移位寄存器

该例中，将每个寄存器的输出作为下一位寄存器的输入，并将高位寄存器的输出作为循环的输入。其 Verilog HDL 程序代码如下：

```
module shiftregist1 (D, clk, reset);
    parameter shiftregist_width = 4;
    output [shiftregist_width-1:0] D;
    input clk, reset;
    reg [shiftregist_width-1:0] D;
       always @(posedge clk)
         if (!reset)
            D <= 4'b0000;
         else
            D <= {D[shiftregist_width-2:0], D[shiftregist_width-1]};
endmodule
```

4.3.4　序列信号发生器

序列信号是数字电路系统中常用的功能单元，其种类很多，如按照序列循环长度 M 与触发器数目 n 的关系，一般可分为三种：

(1) 最大循环长度序列码，$M = 2^n$。

(2) 最长线性序列码(M 序列码)，$M = 2^n - 1$。

(3) 任意循环长度序列码，$M < 2^n$。

序列信号发生器是能够产生一组或多组序列信号的时序电路，它可以由纯时序电路构成，也可以由包含时序逻辑和组合逻辑的混合电路构成。

例 4.3-9　用 Verilog HDL 设计一个产生 100111 序列的信号发生器。

方法 1：由移位寄存器构成。

采用循环移位寄存器，在电路工作前，将所需的序列码置入移位寄存器中，然后循环移位，就可以不断地产生需要的序列。由于移位寄存器输入和输出信号之间没有组合电路，不需要进行组合逻辑的反馈运算，因此这种序列产生电路的工作频率很高。缺点是移位寄存器长度取决于序列长度，因此所占用电路的面积很大。移位寄存器构成的序列信号发生器如图 4.3-13 所示。

其 Verilog HDL 程序代码如下：

```
module signal_maker(out, clk, load, D);
    parameter M = 6;
    output out;
    input clk, load;
    input [M-1:0] D;
    reg [M-1:0] Q;
        initial    Q = 6'b100111;
        always @(posedge clk)
                if (load)    Q <= D;
                        else        Q <= {Q[M-2:0], Q[M-1]};
        assign out = Q[M-1];
endmodule
```

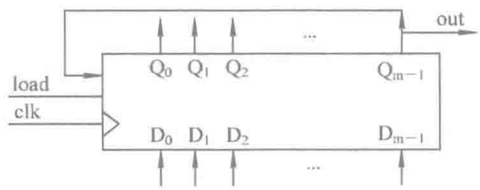


图 4.3-13　移位寄存器构成的序列信号发生器

方法 2：由移位寄存器和组合逻辑电路构成。

反馈移位寄存器型序列信号发生器的结构框图如图 4.3-14 所示，它由移位寄存器和组合反馈网络组成，从移位寄存器的某一输出端可以得到周期性的序列码。

其设计步骤如下：

(1) 根据给定的序列信号的循环周期 M，确定移位寄存器位数 n，$2^{n-1} < M \leqslant 2^n$。

(2) 确定移位寄存器的 M 个独立状态。将给

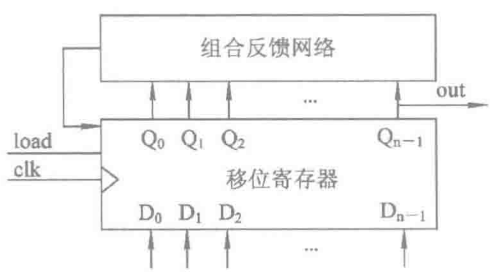


图 4.3-14　反馈移位寄存器型序列信号发生器的结构框图

定的序列码按照移位规律每 n 位一组，划分为 M 个状态。若 M 个状态中出现重复现象，则应增加移位寄存器位数。用 n+1 位再重复上述过程，直至划分为 M 个独立状态。

(3) 根据 M 个不同的状态列出移位寄存器的态序表和反馈函数表，求出反馈函数 F 的表达式。

(4) 检查自启动性能。

与上面的序列信号发生器相比，各个寄存器的输出需经过反馈网络后才连接到移位寄存器的输入端。因此，电路的速度必然下降，但反馈网络的好处在于它可以节省寄存器。

对于“100111”序列的信号发生器，首先确定所需移位寄存器的个数 n。因 $M=6$，故 $n\geq3$。其次确定移位寄存器的 6 个独立状态。按照移位规律每三位一组，划分 6 个状态为 100、001、011、111、111、110。其中状态 111 重复出现，故取 $n=4$，并重新划分状态，得到 1001、0011、0111、1111、1110、1100。因此确定 $n=4$。第三，列态序表和反馈激励函数表，求反馈函数 F 的表达式。先列出态序表，然后根据每一状态所需要的移位输入即反馈输入信号，列出反馈激励函数表(见表 4.3-2)，求得反馈激励函数 $F=\overline{Q}_3+\overline{Q}_1\cdot\overline{Q}_0+Q_3+\overline{Q}_2$。

表 4.3-2 反馈激励函数表

Q_0	Q_1	Q_2	Q_3	F
1	0	0	1	1
0	0	1	1	1
0	1	1	1	1
1	1	1	1	0
1	1	1	0	0
1	1	0	0	1

反馈移位寄存器型序列信号发生器的 Verilog HDL 程序代码如下：

```
module signal_maker(out, clk, load, D);
    parameter M = 4;
    output out;
    input clk, load;
    input [M-1:0] D;
    reg [M-1:0] Q;
        wire w1;
        always @(posedge clk)                              //时序电路部分，移位寄存器
                if (load)    Q <= D;
                else         Q <= {Q[M-2:0], w1};
        assign w1 = (~Q[3])|(~Q[1]&(~Q[0]))|(Q[3]&(~Q[2]));   //组合逻辑电路，反馈网络
        assign out = Q[M-1];
endmodule
```

方法 3：由计数器构成。

计数型序列信号发生器和反馈型序列信号发生器大体相同，都由时序电路和组合电路两部分构成。不同之处在于，反馈型序列信号发生器的时序状态由移位寄存器产生，输出取寄存器的最高位；而在计数型序列信号发生器中，采用计数器代替移位寄存器产生时序状态，输出由组合电路产生。计数型序列信号发生器的优点在于，计数器的状态设置与输出序列没有直接关系，不需要根据输出确定状态，只需要将反馈网络设计好就可以了。因此计数结构对于输出序列的更改比较方便，而且只要连接到不同的反馈网络，就可以同时产生多组序列码。计数型序列信号发生器的结构框图如图 4.3-15 所示。

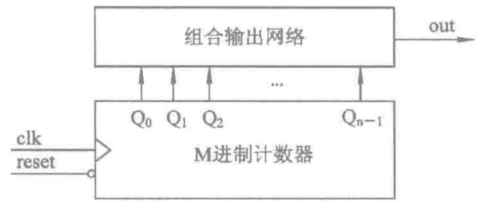


图 4.3-15 计数型序列信号发生器的结构框图

由图可以看出，计数型序列信号发生器和反馈型序列信号发生器大体相似。设计过程分为两步：(1) 根据序列码的长度 M 设计 M 进制计数器，状态可以自定；(2) 按计数器的状态转移关系和序列码的要求设计组合输出网络。

对于“100111”序列的信号发生器，序列信号的 M 值为 6，因为需选用模 6 计数器；计数器的状态选择从 000 到 101；得到的输出组合逻辑真值表见表 4.3-3。

表 4.3-3 输出组合逻辑真值表

Q_2	Q_1	Q_0	out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1

由真值表可画出输出 out 的卡诺图，得到输出函数 $out = Q_2 + \overline{Q}_1\overline{Q}_0 + Q_1Q_0$。其 Verilog HDL 程序代码如下：

```
module signal_maker(out, clk, reset);
    parameter M = 3;
    output out;
    input clk, reset;
    reg [M-1:0] counter;
        always @(posedge clk)
            if (!reset)     counter <= 3'b000;
            else        counter <= counter+1;
        assign out = counter[2]|((~counter[1])&(~counter[0]))|(counter[1]&counter[0]);
endmodule
```

例 4.3-10 用 Verilog HDL 设计伪随机码发生器。

伪随机码是一种变化规律与随机码类似的二进制代码，可以作为数字通信中的一个信

号源，通过信道发送到接收机，用于检测数字通信系统错码的概率，即误码率。

在传统的数字电路设计中，伪随机序列信号发生器是用移位寄存器型计数器来实现的，见图 4.3-14。反馈网络输入信号从移位寄存器的部分输出端(Q_{N-1}～Q_0)中取出，它的输出端 F 反馈到移位寄存器的串行输入端。

通过不同的反馈网络，可以形成不同的移位寄存器型计数器。以 M 序列码为例，反馈函数见表 4.3-4。表中的 N 是触发器的级数，F 是反馈函数。例如 N = 4，则反馈函数如下：

$$F = Q_1 \oplus Q_0 \tag{4.3-10}$$

表 4.3-4　M 序列反馈函数表

N	F	N	F	N	F
1	0	18	5，2，1，0	35	2，0
2	1，0	19	5，2，1，0	36	6，5，4，2，1，0
3	1，0	20	3，0	37	5，4，3，2，1，0
4	1，0	21	2，0	38	6，5，1，0
5	2，0	22	1，0	39	4，0
6	1，0	23	5，0	40	5，4，3，0
7	1，0	24	4，3，1，0	41	3，0
8	4，3，2，0	25	3，0	42	5，4，3，2，1，0
9	4，0	26	6，2，1，0	43	3，2，1，0
10	3，0	27	5，2，1，0	44	6，5，2，0
11	2，0	28	3，0	45	3，2，1，0
12	6，4，1，0	29	2，0	46	10，4，3，2，1，0
13	3，2，1，0	30	6，4，1，0	47	5，0
14	5，3，1，0	31	3，0	48	7，3，2，1，0
15	1，0	32	7，5，3，2，1，0	49	6，5，4，0
16	5，3，2，0	33	6，4，1，0	50	4，3，1，0
17	3，0	34	7，6，5，2，1，0		

下面以 N = 4 为例。在 15 位最长线性序列移位寄存器型计数器中，有一个由“0000”构成的死循环，为了打破死循环，可以修改为

$$F = Q_1 \oplus Q_0 + \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} \tag{4.3-11}$$

当 Q_3～Q_0 = 0000 时，反馈函数 F = 1，打破了原反馈函数 F = 0 出现的死循环。

根据 N = 4 的最长线性序列移位寄存器型计数器的功能，实现的伪随机码发生器的 Verilog HDL 程序代码如下：

```
module signal15(out, clk, load_n, D_load);
    output out;
    input load_n, clk;
    input [3:0] D_load;
    reg [3:0] Q;
```

```
wire F;
    always @(posedge clk)
        if (~load_n)        Q <= D_load;
        else                Q <= {Q[2:0], F};
    assign F = (Q[1]^Q[0])|(~Q[3]&~Q[2]&~Q[1]&~Q[0]);
    assign out = Q[3];
endmodule
```

4.4　有限同步状态机

有限状态机可以分为同步和异步两种，在本书中我们只讨论有限同步状态机，后文中提到的有限状态机均指有限同步状态机。有限状态机是时序电路的通用模型，任何时序电路都可以表示为有限状态机。在由时序电路表示的有限状态机中，各个状态之间的转移总是在时钟的触发下进行的，状态信息存储在寄存器中。因为状态的个数是有限的，所以称为有限状态机。

同其它时序电路一样，有限状态机也由两部分组成：存储电路和组合逻辑电路。存储电路用来生成状态机的状态，组合逻辑电路用来提供输出以及状态机跳转的条件。其电路的结构如图 4.4-1 所示。

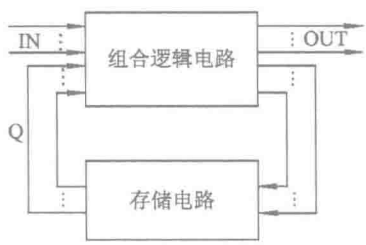


图 4.4-1　有限状态机的电路结构

根据输出信号的产生方式，有限状态机可以分为米利型(Mealy)和摩尔型(Moore)两类。Mealy 型状态机的输出与当前状态和输入有关，Moore 型状态机的输出仅依赖于当前状态，而与输入无关。有限状态机的结构如图 4.4-2 所示。

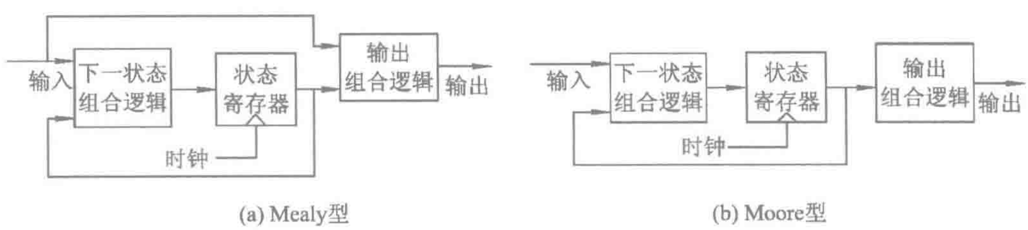


图 4.4-2　有限状态机的结构

状态机的编码方式很多，由此产生的电路也不相同。常见的编码方式有三种：二进制编码、格雷编码和一位独热编码。

(1) 二进制编码：其状态寄存器是由触发器组成的。N 个触发器可以构成 2^n 个状态。二进制编码的优点是使用的触发器个数较少，节省了资源；缺点是状态跳转时可能有多个 bit(位)同时变化，引起毛刺，造成逻辑错误。

(2) 格雷编码：与二进制编码类似。格雷编码状态跳转时只有一个 bit(位)发生变化，减少了产生毛刺和一些暂态的可能。

(3) 一位独热(One Hot)编码：这是对于 n 个状态采用 n 个 bit(位)来编码，每个状态编码中只有一个 bit(位)为 1，如 0001、0010、0100、1000。One Hot 编码增加了使用触发器的个数，但该种编码方式为以后的译码提供了方便，能有效地节省和简化组合电路。

在 Verilog HDL 中，有限状态机的描述方法较多，常用的有两段式和三段式两种。

(1) 状态机的两段式描述方法：

```
//第一个进程，同步时序 always 模块，格式化描述次态寄存器迁移到现态寄存器
always @(posedge clk or negedge rst_n)          //异步复位
    if(!rst_n)    current_state <= IDLE;
    else   current_state <= next_state;          //注意，使用的是非阻塞赋值
//第二个进程，组合逻辑 always 模块，描述状态转移条件判断
always @(current_state or  输入信号)            //电平触发
  begin
   next_state = x;                 //要初始化，使得系统复位后能进入正确的状态
     case(current_state)
       S1: if(...)
       next_state = S2;            //阻塞赋值
       out1 <= 1'b1;               //注意是非阻塞逻辑
   ...
     endcase
  end
```

(2) 状态机的三段式描述方法：

```
//第一个进程，同步时序 always 模块，格式化描述次态寄存器迁移到现态寄存器
always @(posedge clk or negedge rst_n)          //异步复位
    if(!rst_n) current_state <= IDLE;
    else current_state <= next_state;            //注意，使用的是非阻塞赋值
//第二个进程，组合逻辑 always 模块，描述状态转移条件判断
always @(current_state or  输入信号)       //电平触发
  begin
     next_state = x;                      //要初始化，使得系统复位后能进入正确的状态
     case(current_state)
       S1: if(...)
       next_state = S2;                   //阻塞赋值
    ...
      endcase
```

```
  end
 //第三个进程，同步时序 always 模块，格式化描述次态寄存器输出
always @ (posedge clk or negedge rst_n)
begin
...//初始化
   case(next_state or  输入信号)
        S1:
        out1 <= 1'b1;                        //注意是非阻塞逻辑
        S2:
        out1 <= 1'b0;
        default:...                          //default 的作用是免除综合工具综合出锁存器
  endcase
end
```

三段式并不是一定要写三个 always 块，如果状态机更为复杂，always 块也会相应增加。

例 4.4-1 用 Verilog HDL 设计顺序脉冲发生器。

顺序脉冲发生器又称脉冲分配器，它将高电平脉冲依次分配到不同的输出上，保证在每个时钟周期内只有一路输出高电平，不同时钟上的高电平脉冲依次出现在所有输出端。

以 4 位顺序脉冲发生器为例，它有 4 路输出 S_0、S_1、S_2、S_3，每路输出上高电平脉冲依次出现，输出在 1000、0100、0010、0001 之间循环。4 位顺序脉冲发生器的状态转移图如图 4.4-3 所示，它由 4 个状态构成，每个状态中“1”的个数都是 1 个，表示每个时钟周期内只有一路输出端为高电平，而且是轮流出现，因此生成了顺序脉冲信号。

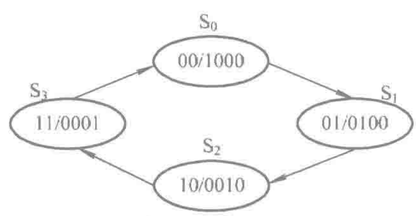


图 4.4-3　4 位顺序脉冲发生器电路的状态转移图

对四状态的状态机编码只需要两位二进制编码即可，Verilog HDL 程序设计代码如下：

```
module state4(OUT, clk, rst_n);
    output [3:0] OUT;
    input clk;
    input rst_n;
    reg [3:0] OUT;
    reg [1:0] STATE, next_STATE;
    always @(STATE)
      case (STATE)
        2'b00:
            begin
              OUT <= 4'b1000;
              next_STATE <= 2'b01;
            end
        2'b01:
            begin
```

```
            OUT <= 4'b0100;
            next_STATE <= 2'b10;
        end
    2'b10:
        begin
            OUT <= 4'b0010;
            next_STATE <= 2'b11;
        end
    2'b11:
        begin
            OUT <= 4'b0001;
            next_STATE <= 2'b00;
        end
    endcase
  always@(posedge clk or negedge rst_n)
    if (!rst_n) STATE <= 2'b00;
    else STATE <= next_STATE;
endmodule
```

例 4.4-2 设计一个自动售报机，报纸价钱为八角，纸币有 1 角、2 角、5 角、一元。该自动售报机不考虑投币为大额面值等特殊情况。

图 4.4-4 是自动售报机的状态转移图。图中，S_0～S_7 为状态机的 8 个状态，角标代表已投币的总和，如 S_0 代表没有投币，S_1 代表已投入 1 角，依此类推。M 代表输入，M1 表示投入 1 角硬币，M2 代表投入 2 角硬币，M5 代表投入 5 角硬币，M10 代表投入一元。

data_out = 1 表示给出报纸，data_out_return1 = 1 表示找回 1 角硬币，data_out_return2 = 1 表示找回 2 角硬币。自动售报机的 Verilog HDL 程序代码如下：

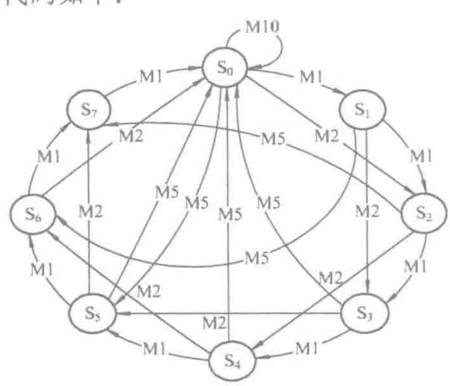


图 4.4-4 自动售报机的状态转移图

```
module auto_sellor(current_state, data_out,
      data_out_return1, data_out_return2, clk,
      rst_n, data_in);
   parameter state_width = 3, data_in_width = 3;
   output [state_width-1:0] current_state;
   output data_out, data_out_return1,
         data_out_return2;
   input [data_in_width-1:0] data_in;
   input clk, rst_n;
   reg [state_width-1:0] current_state, next_state;
   reg data_out, data_out_return1, data_out_return2;
   always @(current_state or data_in)
       case (current_state)
         3'b000: case (data_in)
```

```
            3'b000: begin
                    next_state <= 3'b000;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
            3'b001: begin
                    next_state <= 3'b001;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
            3'b010: begin
                    next_state <= 3'b010;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
            3'b011: begin
                    next_state <= 3'b101;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
            3'b100: begin
                    next_state <= 3'b000;
                    data_out <= 1'b1;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b1;
                  end
          endcase
3'b001: case (data_in)
            3'b000: begin
                    next_state <= 3'b001;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
            3'b001: begin
                    next_state <= 3'b010;
```

```
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
          3'b010: begin
                    next_state <= 3'b011;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
          3'b011: begin
                    next_state <= 3'b110;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
        endcase
3'b010: case (data_in)
          3'b000: begin
                    next_state <= 3'b010;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
          3'b001: begin
                    next_state <= 3'b011;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
          3'b010: begin
                    next_state <= 3'b100;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                  end
          3'b011: begin
                    next_state <= 3'b111;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
```

```
                    data_out_return2 <= 1'b0;
                end
            endcase
    3'b011: case (data_in)
                3'b000: begin
                    next_state <= 3'b011;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
                3'b001: begin
                    next_state <= 3'b100;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
                3'b010: begin
                    next_state <= 3'b101;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
                3'b011: begin
                    next_state <= 3'b000;
                    data_out <= 1'b1;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
            endcase
    3'b100: case (data_in)
                3'b000: begin
                    next_state <= 3'b000;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
                3'b001: begin
                    next_state <= 3'b101;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
```

```
                    data_out_return2 <= 1'b0;
                end
            3'b010: begin
                    next_state <= 3'b110;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
            3'b011: begin
                    next_state <= 3'b000;
                    data_out <= 1'b1;
                    data_out_return1 <= 1'b1;
                    data_out_return2 <= 1'b0;
                end
        endcase
    3'b101: case (data_in)
            3'b000: begin
                    next_state <= 3'b101;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
            3'b001: begin
                    next_state <= 3'b110;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
            3'b010: begin
                    next_state <= 3'b111;
                    data_out <= 1'b0;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b0;
                end
            3'b011: begin
                    next_state <= 3'b000;
                    data_out <= 1'b1;
                    data_out_return1 <= 1'b0;
                    data_out_return2 <= 1'b1;
                end
```

```
            endcase
        3'b110: case (data_in)
                3'b000: begin
                        next_state <= 3'b110;
                        data_out <= 1'b0;
                        data_out_return1 <= 1'b0;
                        data_out_return2 <= 1'b0;
                    end
                3'b001: begin
                        next_state <= 3'b111;
                        data_out <= 1'b0;
                        data_out_return1 <= 1'b0;
                        data_out_return2 <= 1'b0;
                    end
                3'b010: begin
                        next_state <= 3'b000;
                        data_out <= 1'b1;
                        data_out_return1 <= 1'b0;
                        data_out_return2 <= 1'b0;
                    end
            endcase
        3'b111: case (data_in)
                3'b000: begin
                        next_state <= 3'b111;
                        data_out <= 1'b0;
                        data_out_return1 <= 1'b0;
                        data_out_return2 <= 1'b0;
                    end
                3'b001: begin
                        next_state <= 3'b000;
                        data_out <= 1'b1;
                        data_out_return1 <= 1'b0;
                        data_out_return2 <= 1'b0;
                    end
            endcase
    endcase

always @(posedge clk or rst_n)
    if (!rst_n)
        current_state <= 3'b000;
```

```
    else
        current_state <= next_state;


endmodule
```

例 4.4-3 “11010”序列检测器示例。

序列检测器就是将一个指定的序列从数字码流中检测出来。当输入端出现序列 11010 时，输出为 1，否则输出为 0。在此不考虑重复序列，即出现指定序列后就重新开始序列检测，不再考虑以前的数据。规定数据从右端输入，即按照 1—1—0—1—0 的顺序输入。该序列检测器的状态转移图如图 4.4-5 所示。

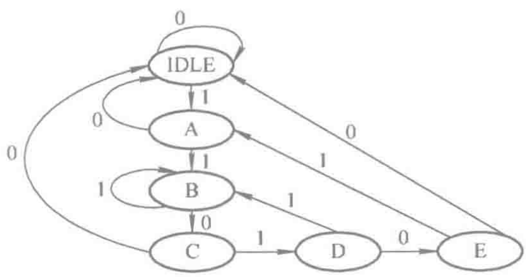


图 4.4-5 “11010”序列检测器的状态转移图

其 Verilog HDL 程序代码如下：

```
module seqdet (D_out, D_in, rst_n, clk);
    parameter IDLE = 3'd0, A = 3'd1, B = 3'd2, C = 3'd3, D = 3'd4, E = 3'd5;
    output D_out;
    input D_in, rst_n, clk;
    reg [2:0] state, next_state;
    wire D_out;
        assign D_out = (state = = E)?1:0;
        always @(state or D_in)
            case (state)
              IDLE : if (D_in) next_state = A;
                        else next_state = IDLE;
              A : if (D_in) next_state = B;
                  else next_state = IDLE;
              B : if (D_in) next_state = B;
                  else next_state = C;
              C : if (D_in) next_state = D;
                  else next_state = IDLE;
              D : if (D_in) next_state = B;
                  else next_state = E;
              E : if (D_in) next_state = IDLE;
                  else next_state = A;
```

```
            default : next_state = IDLE;
          endcase
      always @(posedge clk)
        state <= next_state;
endmodule
```

本 章 小 结

本章先介绍了 Verilog HDL 的可综合性，随后介绍了四部分内容，即组合电路、时序电路、混合电路以及有限同步状态机。硬件描述语言的基础在于对电路的认识。本章实质上介绍的是电路的 Verilog HDL 设计方法，包含加法器、数据选择器等，可为以后的代码编写打下硬件基础。

思考题和习题

1. Verilog HDL 的设计方法有哪三种？
2. Verilog HDL 模块的结构描述方式可以分为哪几种？
3. 逻辑变量的取值 1 和 0 可以分别表示电平和逻辑的什么？
4. 当逻辑函数有 n 个变量时，共有多少个变量取值组合？
5. 假设编码器中有 50 个编码对象，请输出二进制代码位数？
6. 一个 16 选 1 的数据选择器，其地址输入(选择控制输入)端有多少个？
7. N 个触发器可以构成能寄存多少位二进制数码的寄存器？
8. 写出 8 bit 加法器的逻辑表达式，比较用超前进位逻辑和不用超前进位逻辑的电路速度。
9. 利用数字电路的基本知识，解释：即使组合逻辑电路的输入端的所有信号同时变化，其输出端的各个信号也不可能同时达到新的值；各个信号变化快慢的决定因素。
10. 提高复杂组合逻辑运算速度有哪些办法？
11. 在设计加法器时，除了使用基本门电路结构描述方式外，是否还可以利用算数操作符实现？写一个加法器，用综合工具综合。
12. 为什么采用流水线的办法可以显著提高层次多的复杂组合逻辑的运算速度？
13. 使用状态机设计电路时，不同的状态分配对电路的复杂度和速度有何影响？
14. 根据以下程序，画出所产生的信号 a、c 的波形。

```
initial
  begin
    a <= 0;
    b <= 1;
    c <= 0;
    d <= 1;
    #5  a <= b;
```

```
    #10 c <= d;
  end
```

15. 请分析以下两段 Verilog HDL 程序所描述电路的区别。

程序(1)：

```
module bloc(clk, a, b);
    input clk, a;
    output b;
        reg y;
        reg b;
        always @(posedge clk)
          begin
            y = a;
            b = y;
          end
endmodule
```

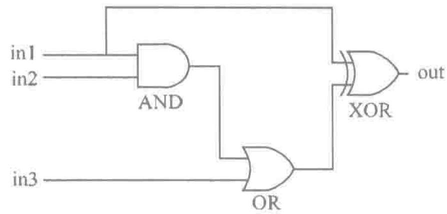
程序(2)：

```
module nonbloc(clk, a, b);
    input clk, a;
    output b;
    reg y;
    reg b;
    always @(posedge clk)
      begin
        y <= a;
            b <= y;
      end
endmodule
```

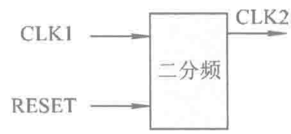
16. 试用查找表的方式实现真值表中的加法器，并写出 Verilog HDL 代码。

Cin	ain	bin	sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

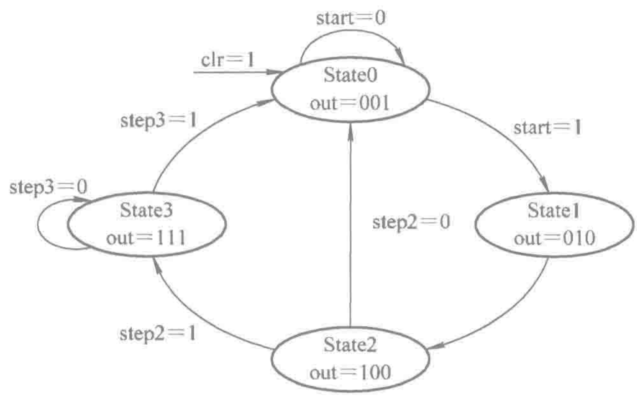
17. 试用 Verilog HDL，利用内置基本门级元件，以结构描述方式生成下图所示的电路。



18. 试用 Verilog HDL 描述用 D 触发器实现带有同步清零功能(低电平有效)的二分频的逻辑电路，并设计功能模块的测试程序(见下图)。



19. 用 Verilog HDL 描述下面的状态转移图。



20. 用状态机的方式设计一个序列检测器，当检测到连续 4 个或 4 个以上的“1”输入时，输出为“1”，其它输入情况下的输出为“0”。画出状态转移图，写出二段式 Verilog HDL 代码。

21. NRZ 码(非归零码)是一种最简单、最常用的信号形式。该信号由正电平对应二进制代码 1，零电平对应二进制代码 0。将 NRZ 码的 1 用 10 表示、0 用 01 表示，即得到 Manchester(曼彻斯特)码。设计一个 Mealy 型有限状态机，将 NRZ 码比特流转换成 Manchester 码比特流。

第 5 章 仿真验证与 Testbench 编写

5.1 Verilog HDL 电路仿真和验证概述

在 Verilog HDL 集成电路设计过程中，设计者完成 RTL 级描述后需要对设计进行设计确认。设计确认是设计者检查设计中是否包含缺陷的过程。在设计中，表述不清的设计规范、设计者的错误或者错误地调用了元件等都可能给设计带来缺陷。因此，设计确认对于集成电路设计来说具有重要的作用。设计确认可以通过仿真和验证来完成。仿真和验证能确保设计的完整性、可靠性、实效性以及先进性。

仿真，也可以称为模拟，是通过 EDA 仿真工具，对所设计电路或系统输入测试信号，然后根据其输出信号(波形、文本或者 VCD 文件)与期望值进行比较，来确认是否得到与所期望一致的设计结果，从而验证设计的正确性。

在设计过程中，仿真是在综合之前完成的，这就是通常所说的行为级仿真、RTL 仿真或前仿真。在 RTL 设计阶段只包含了时钟及其时序，并未包含门延时和线延时。因此，RTL 仿真对于时钟来说是正确的，并且不用考虑竞争冒险、毛刺、建立和保持时间以及其它一些详细的时序问题，这样 RTL 仿真就具有较快的速度。

验证是一系列测试平台的集合，是一个证明设计思路如何实现及保证设计在功能上正确的过程。验证在 Verilog HDL 设计的整个流程中分为 4 个阶段：阶段 1——功能验证；阶段 2——综合后验证；阶段 3——时序验证；阶段 4——板级验证。其中前 3 个阶段是在 PC 平台上依靠 EDA 工具来实现的，最后一个阶段则需要在真正的硬件平台(FPGA、CPLD 等)上进行，需要借助一些调试工具或者专业性的分析仪来调试，因此本书所介绍的验证仅限于在 PC 平台上运行的前 3 个阶段的验证。

在测试验证环节中，要求测试需具备高效、完备的特性。高效是指以最短的时间发现错误，从而能以最短的时间上市；完备是指发现全部的错误，要求测试需达到一定的覆盖率，包括代码的覆盖率和功能的覆盖率。

目前常用的功能验证有三种：黑盒法、白盒法和灰盒法。

1) 黑盒法

黑盒法就是把测试代码看做一个黑盒子，测试人员完全不考虑代码内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。验证人员在 RTL 级输入端施加激励信号，然后将输出值与期望值相比较，以验证设计的正确性。

黑盒法主要有两个优点：① 简单。验证者无需了解 RTL 级设计的细节，只需根据规格说明书搭建 Testbench(测试平台)。② 易于实现验证和设计的独立性。由于验证者不了解 RTL

级设计细节，在搭建 Testbench 时不会受设计者思路的影响，因此能避免按 RTL 级设计者的实现思路来验证 RTL 级设计的情况。

黑盒法的主要缺点是可观测性差。由于验证人员对内部的实现细节不了解，无法插入内部测试点，因此很难对错误进行迅速定位，在大规模设计中难以跟踪错误的根源。黑盒法一般适用于中小规模电路的验证。

2) 白盒法

白盒法也称结构测试或逻辑驱动测试，它是按照 RTL 级代码内部结构进行测试，通过测试来检测 RTL 级代码内部实现是否按照设计规格说明书的规定正常执行，检验 RTL 级代码中的每条路径是否都能按预定要求正确工作。

验证人员是在对内部的设计细节熟悉且能够对内部信号完全控制和观察的情况下进行验证的。

白盒法的优点在于容易观察和控制验证的进展状况，可以根据事先设置的观测点，在错误出现后很快定位问题的根源。其缺点则是需要耗费很长的时间去了解 RTL 级的实现细节，且难以实现设计与验证的分离，验证团队可能会受设计团队思路的影响，出现沿着设计思路去验证的现象，结果是无法证明设计的功能是否正确。

3) 灰盒法

灰盒法是介于黑盒法和白盒法之间的一种测试方法。灰盒测试关注输出对于输入的正确性，同时也关注内部表现，但这种关注没有白盒法那样详细、完整，只是通过一些表征性的现象、事件、标志来判断内部的运行状况。在很多测试中经常会出现输出正确、内部错误的情况，如果每次都通过白盒测试来操作，效率会很低，因此可采取灰盒法。

5.2　Verilog HDL 测试程序设计基础

5.2.1　Testbench 及其结构

在 Verilog HDL 中，通常采用测试平台(Testbench)方式进行仿真和验证。在仿真时，Testbench 用来产生测试激励给待验证设计(Design Under Verification，DUV)，或者称为待测试设计(Design Under Test，DUT)，同时检查 DUV/DUT 的输出是否与预期的一致，从而达到验证设计功能的目的，如图 5.2-1 所示。

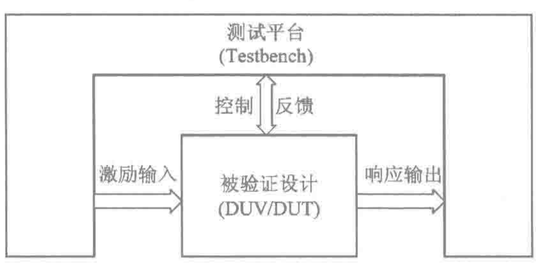


图 5.2-1　Testbench 结构

应该指出的是，由于 Testbench 是一个测试平台，信号集成在模块内部，因此没有输入、输出。在 Testbench 模块内，例化待测设计的顶层模块，并把测试行为的代码封装在内，直

接对待测系统提供测试激励。图 5.2-2 给出了一个典型的 Testbench 程序结构。



图 5.2-2　典型的 Testbench 程序结构

下面举例介绍测试程序中各模块的分布。

例 5.2-1　T 触发器测试程序示例。

```
module Tflipflop_tb;
    //数据类型声明
    reg clk, rst_n, T;
    wire data_out;
    //对被测试模块实例化
    TFF U1 (.data_out(data_out), .T(T), .clk(clk), .rst_n(rst_n));
    //产生测试激励
        always
          #5 clk = ~clk;
        initial
          begin
            clk = 0;
            #3 rst_n = 0;
            #5 rst_n = 1;
            T = 1;
            #30   T = 0;
            #20   T = 1;
       end
    //对输出响应进行收集
        initial
          begin
            $monitor($time, "T = %b, clk = %b, rst_n = %b,
```

```
            data_out = %b", T, clk, rst_n, data_out);
      end
endmodule
```

T 触发器的仿真波形如图 5.2-3 所示。

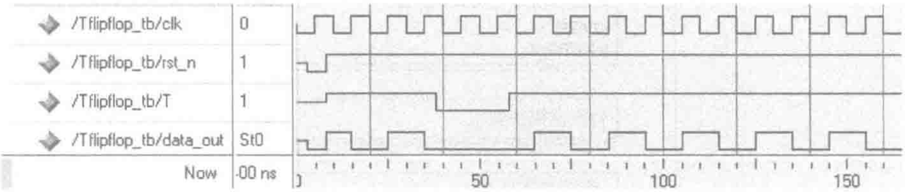


图 5.2-3　T 触发器的仿真波形

部分文本输出结果：

```
0T = x, clk = 0, rst_n = x, data_out = x
3T = x, clk = 0, rst_n = 0, data_out = 0
5T = x, clk = 1, rst_n = 0, data_out = 0
8T = 1, clk = 1, rst_n = 1, data_out = 1
10T = 1, clk = 0, rst_n = 1, data_out = 1
```

Testbench 模块最重要的任务就是利用各种合法的语句，产生适当的时序和数据，以完成测试，并达到覆盖率要求。一般来讲，测试激励信号在 initial 语句块和 always 语句块中进行赋值，因此，与被测试模块的输入端口相连的输入激励信号定义为 reg 类型；与被测试模块输出端口相连的信号定义为 wire 类型，主要用于对测量结果的观察。

仿真因 EDA 工具和设计复杂度的不同而略有不同，对于简单的设计，特别是一些小规模的数字设计来说，一般可以直接使用开发工具内嵌的仿真波形工具绘制激励信号，然后进行功能仿真。另外一种较为常用的方式是，使用 HDL(硬件描述语言)编制 Testbench(仿真文件)，通过波形或自动比较工具，分析设计的正确性，并分析 Testbench 自身的覆盖率和正确性。

基于 Testbench 的仿真流程如图 5.2-4 所示。从图中可以清晰地看出 Testbench 的主要功能：

(1) 为 DUT 提供激励信号；

(2) 正确实例化 DUT；

(3) 将仿真数据显示在终端或者存为文件，也可以显示在波形窗口中供分析检查；

(4) 复杂设计可以使用 EDA 工具，或者通过用户接口自动比较仿真结果与理想值，实现结果的自动检查。

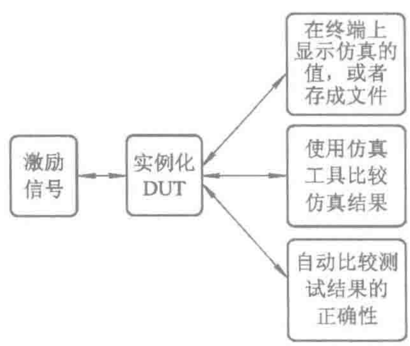


图 5.2-4　基于 Testbench 的仿真流程

前两点功能主要和 Testbench 的编写方法和风格(Coding Style)相关，后两点功能主要和仿真工具的功能特性及支持的用户接口相关。仿真工具将在后面章节中详细介绍，本章所要介绍的重点是如何编写规范、高效、合理的测试程序。

Testbench 设计好以后，可以为芯片设计的各个阶段服务。比如，对 RTL 代码、综合网表和布线之后的网表进行仿真时，都可以采用同一个 Testbench。

编写 Testbench 时需要注意的问题：

(1) Testbench 代码不需要可综合性。Testbench 代码只是硬件行为描述而不是硬件设计。Testbench 只用于仿真软件中模拟硬件功能，不会被实现成电路，不需要具备可综合性。因此，在编写 Testbench 的时候，要尽量使用抽象层次较高的语句，不仅具备高的代码书写效率，而且准确、仿真效率高。

(2) 行为级描述效率高。如前所述，Verilog HDL 具备 5 个描述层次，分别为开关级、门级、RTL 级、算法级和系统级。虽然所有的 Verilog HDL 都可用于 Testbench 中，但是行为级描述代码具有显著优势：① 降低了测试代码的书写难度，测试人员不需要理解电路的结构和实现方式，从而节约了测试代码的开发时间。② 便于根据需要从不同层次进行抽象设计。在高层次描述中，设计更加简单、高效，只有在需要解析某个模块的详细结构时，才会使用低层次的详细描述。③ EDA 工具本身就支持 Testbench 中的高级数据结构和运算，其编译、运行和行为级仿真速度较快。

(3) 掌握结构化、程式化的描述方式。结构化的描述有利于设计维护。由于在 Testbench 中，所有的 initial、always 以及 assign 语句都是同时执行的，其中每个描述事件都是基于时间“0”点开始的，因此可通过这些语句将不同的测试激励划分开来。一般不要将所有的测试都放在一个语句块中。

5.2.2 测试平台举例

为了验证设计模块功能的正确性，需要在 Testbench 中编写信号激励给设计模块，同时观察这些激励在设计模块中的响应是否与设计目标值一致。

图 5.2-5 中给出了一个 DUT 的仿真平台。测试平台需要产生时钟信号、复位信号和一系列的仿真向量，观察 DUT 的响应，确认仿真结果。

建立 Testbench 进行仿真的流程分为编写仿真激励、搭建仿真环境、确认仿真结果三个步骤。为一个设计建立仿真平台，将这个设计在该平台中实例化，然后将在平台中产生的测试激励输入给设计模块，再观察 DUT 的响应是否与期望值相同。下面介绍组合逻辑电路和时序逻辑电路测试仿真环境的搭建。

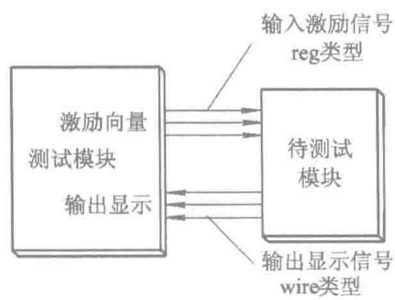


图 5.2-5　DUT 仿真平台

1. 组合逻辑电路仿真环境的搭建

组合逻辑电路的设计验证，主要是检查设计结果是否符合该电路真值表的功能，因此在搭建组合逻辑电路仿真环境时，用 initial 语句块把被测电路的输入按照真值表提供的数据变化作为测试条件。组合逻辑电路的特点决定了仿真中只需对输入信号进行设计即可，没有时序、定时信息和全局复位、置位等信号要求。

例 5.2-2　搭建全加器的仿真环境。

全加器的真值表如表 5.2-1 所示。

表 5.2-1　全加器的真值表

a	b	ci	so	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

用 Verilog HDL 编写的全加器程序代码如下：

```
module adder1(a, b, ci, so, co);
    input   a, b, ci;
    output  so, co;
    assign{co, so} = a+b+ci;
endmodule
```

根据全加器的真值表(表 5.2-1)编写的全加器测试程序如下：

```
module adder1_tb;
    wire so, co;
    reg   a, b, ci;
    adder1 U1(a, b, ci, so, co);      //模块实例化
    initial                           //测试信号产生
      begin
              a=0; b=0; ci=0;
          #20 a=0; b=0; ci=1;
          #20 a=0; b=1; ci=0;
          #20 a=0; b=1; ci=1;
          #20 a=1; b=0; ci=0;
          #20 a=1; b=0; ci=1;
          #20 a=1; b=1; ci=0;
          #20 a=1; b=1; ci = 1;
          #200   $finish;
      end
   endmodule
```

在源程序中，把全加器的输入 a、b 和 ci 定义为 reg 型变量；把输出 so 和 co 定义为 wire 型变量；用模块实例化语句“adder1 U1(a, b, ci, so, co);”把全加器设计电路例化到测试仿真环境中；用 initial 语句块改变输入的变化并生成测试条件，输入的变化语句完全根据全加器的真值表编写。其仿真结果如图 5.2-6 所示。

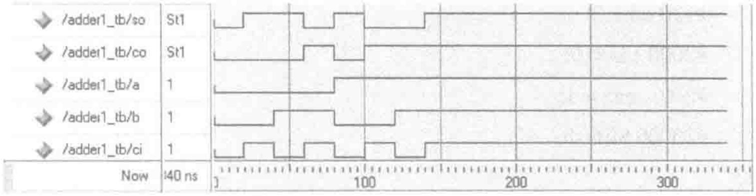


图 5.2-6　全加器的仿真结果

2. 时序逻辑电路仿真环境的搭建

时序逻辑电路仿真环境的搭建要求与组合逻辑电路基本相同，主要区别在于时序逻辑电路仿真环境中，需要考虑时序、定时信息和全局复位、置位等信号要求，并定义这些信号。

例 5.2-3　搭建十进制加法计数器的仿真环境。

用 Verilog HDL 编写的十进制加法计数器的源程序代码如下：

```
module cnt10(clk, rst, ena, q, cout);
    input     clk, rst, ena;
    output    [3:0]   q;
    output     cout;
    reg       [3:0] q;
    always @ (posedge clk or posedge rst)
        begin
            if(rst)q = 4'b0000;
            else if(ena)
                begin
                    if(q<9)q = q+1;
                    else   q = 0;
                end
           end
           assign cout = q[3]&q[0];
  endmodule
```

Verilog HDL 测试程序代码如下：

```
module cnt10_tb;
    reg       clk, rst, ena;
    wire   [3:0] q;
    wire   cout;
    cnt10 U1(clk, rst, ena, q, cout);              //模块实例化
    always          #50   clk = ~clk;      //时钟信号产生
    initial
      begin
           clk = 0; rst = 0; ena = 1;              //控制信号产生
           #1200 rst = 1;
```

```
            #120 rst = 0;
            #2000 ena = 0;
            #200    ena = 1;
            #20000 $finish;
        end
    endmodule
```

在源程序中，用模块实例化语句“cnt10 U1(clk, rst, ena, q, cout); ”把十进制计数模块例化到仿真环境中；在 always 中用语句“#50 clk = ~clk; ”产生周期为 100(标准时间单位)的时钟方波；用 initial 语句块生成复位信号 rst 和使能控制信号 ena 的测试条件。其仿真结果如图 5.2-7 所示。

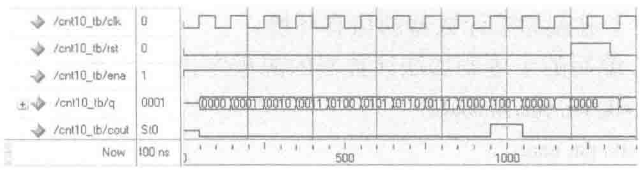


图 5.2-7　十进制加法计数器的仿真结果

5.2.3 Verilog HDL 仿真结果确认

Verilog HDL 仿真结果可以通过观察波形、观察文本输出、自动检查仿真结果和使用 VCD 文件等方式来确认其正确性。

1) 观察波形

该方式即通过直接观察各信号波形的输出，比较测试值和期望值的大小，来确定仿真结果的正确性。图 5.2-8 是 ModelSim 信号波形观察窗口。这种方法的优势在于观测信号比较直观，适用于观测时间较短的测试信号。

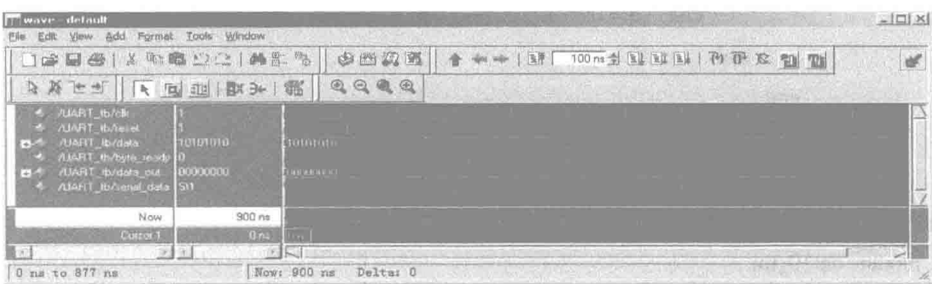


图 5.2-8　ModelSim 信号波形观察窗口

2) 观察文本输出

Verilog HDL 语法中规定了一系列系统任务，其中的打印任务主要用来协助查看仿真结果。例如，$display，即直接输出到标准输出设备；$monitor，即监控参数的变化；$fdisplay，即输出到文件；等等。下面的程序是在例 5.2-2 的测试平台中增加了用于屏幕打印的任务 $monitor。

```
module adder1_tb;
    wire so, co;
```

```
reg   a, b, ci;
adder1 U1(a, b, ci, so, co);          //模块实例化
initial                               //测试信号产生
   begin
          a = 0; b = 0; ci = 0;
      #20 a = 0; b = 0; ci = 1;
      #20 a = 0; b = 1; ci = 0;
      #20 a = 0; b = 1; ci = 1;
      #20 a = 1; b = 0; ci = 0;
      #20 a = 1; b = 0; ci = 1;
      #20 a = 1; b = 1; ci = 0;
      #20 a = 1; b = 1; ci = 1;
      #200   $finish;
   end
  initial   $monitor($time, "%b %b %b -> %b %b", a, b, ci, so, co);
endmodule
```

其输出的结果是：

```
0   0 0 0 -> 0 0
20  0 0 1 -> 1 0
40  0 1 0 -> 1 0
60  0 1 1 -> 0 1
80  1 0 0 -> 1 0
```

3) 自动检查仿真结果

自动检查仿真结果是通过在设计代码中的关键节点添加断言监控器，形成对电路逻辑综合的注释或是对设计特点的说明，以提高设计模块的可观察性。在行为约束时观察设计的关键节点是否达到预期值，如果断言监控器检查到信号没有达到其预期值，则产生相应的信息及差异的时间，这类信息通常出现在仿真报告、脚本或者控制信息里。

4) 使用 VCD 文件

Verilog HDL 提供了一系列系统任务，用于记录信号值变化并保存到标准的 VCD(Value Change Dump)格式数据库中。VCD 文件是一种标准格式的波形记录文件，只记录发生变化的波形。可以将仿真器中的仿真结果输出成一个 VCD 文件，然后将该 VCD 文件输入给其它第三方分析工具进行分析。VCD 文件将在第 5.3.7 节中详细讲述。

5.2.4 Verilog HDL 仿真效率

和 C/C++ 等软件语言相比，Verilog HDL 行为级仿真代码的执行时间比较长，其主要原因就是要通过串行软件代码完成并行语义的转化。随着代码的增加，会使得仿真验证过程非常漫长，从而导致仿真效率的降低，这成为整体设计的瓶颈。即便如此，不同的设计代码其仿真执行效率也是不同的。下面几点可以帮助设计人员提高 Verilog HDL 代码的仿真执行时间。

1. 减小层次结构

仿真代码的层次越少，执行时间就越短。这主要是因为参数在模块端口之间传递时需要消耗仿真器的执行时间。

2. 减少门级代码的使用

由于门级建模属于结构化建模，自身参数建模较复杂，需要通过建模调用的方式来实现，因此建议仿真代码尽量使用行为级语句，建模层次越抽象，执行时间就越短。

3. 仿真精度越高，效率越低

Verilog HDL 语法中提供了计时单位值和计时精度值，并且要求计时单位值大于等于计时精度值。由于在 Verilog HDL 模型中，所有时延都是通过单位时间进行描述的，并且是一个相对的概念，所以计时单位值与计时精度值的差距越大，则模拟时间越长。例如包含`timescale 1ns/1ps 定义的代码执行时间就比包含`timescale 1ns/1ns 定义的代码执行时间长。因此在仿真中，需要通过具体的仿真要求，设定合适的仿真时间和分辨率，提高仿真效率。`timescale 仿真时间标度将在第 5.9.3 节中详细讲述。

4. 进程越少，效率越高

代码中的语句块越少，仿真越快，例如将相同的逻辑功能分布在两个 always 语句块中，其仿真执行时间就比利用一个 always 语句来实现的代码长。这是因为仿真器在不同进程之间进行切换也需要时间。

5. 减少仿真器的输出显示

Verilog HDL 中包含一些系统任务，可以在仿真器的控制台显示窗口输出一些提示信息。虽然它对于软件调试是非常有用的，但会降低仿真器的执行效率，因此在代码中，这一类系统任务不能随意使用。

从本质上来讲，减少代码执行时间并不一定会提高代码的验证效率，因此上述建议需要和代码的可读性、可维护性以及验证覆盖率等方面结合起来考虑。

5.3　与仿真相关的系统任务

5.3.1　$display 和$write

Verilog HDL 中有两种主要的标准输出任务：$display 和 $write。这两个系统函数都用于输出信息且语法格式相同，其语法格式分别如下：

```
$display("<format_specifiers>", <signal1, signal2, …, signaln>);
$write("<format_specifiers>", <signal1, signal2, …, signaln>);
```

其中，“<format_specifiers>”通常称为“格式控制”，而“<signal1, signal2, …, signaln>”则为“信号输出列表”。

虽然 $display 和 $write 的作用相同，但$display 是将特定信息输出到标准输出设备，并且带有行结束字符，即自动地在输出后进行换行；而$write 输出特定信息时不自动换行。如果想在一行里输出多个信息，可以使用 $write。

输出格式说明由“%”和格式字符组成，其作用是将输出的数据转换成指定的格式输出。格式说明总是由“%”字符开始的。对于不同类型的数据用不同的格式输出。表 5.3-1 中给出了常用的几种输出格式，无论何种格式都以较短的结果输出。

表 5.3-1 常用的输出格式

输出格式	说 明	输出格式	说 明
%h 或 %H	以十六进制数的形式输出	%m 或 %M	输出等级层次的名字
%d 或 %D	以十进制数的形式输出	%s 或 %S	以字符串的形式输出
%o 或 %O	以八进制数的形式输出	%t 或 %T	以当前的时间格式输出
%b 或 %B	以二进制数的形式输出	%e 或 %E	以指数的形式输出实型数
%c 或 %C	以 ASCII 码字符的形式输出	%f 或 %F	以十进制数的形式输出实型数
%v 或 %V	输出网络型数据信号强度	%g 或 %G	以指数或十进制数的形式输出实型数

如果需要原样输出的字符，即普通字符，其中一些特殊的字符可以通过表 5.3-2 中的转换序列来输出。表中的字符形式用于格式字符串参数中，用来显示特殊的字符。

表 5.3-2 特殊字符的输出格式

换码序列	功 能
\n	换行
\t	横向跳格(即跳到下一个输出区)
\\	反斜杠字符\
\"	双引号字符"
\o	1 到 3 位八进制数代表的字符
%%	百分符号%

在 $display 和 $write 的参数列表中，其“信号输出表列”是需要输出的一些数据，可以是表达式。下面举例说明。

例 5.3-1 $display 语句示例。

```
module disp_tb;
    reg[31:0] rval;
    pulldown(pd);
        initial
        begin
            rval = 101;
            $display("\\\t%%\n\"\123");                    //八进制数 123 就是字符 S
            $display("rval = %h hex %d decimal", rval, rval);
            $display("rval = %o otal %b binary", rval, rval);
            $display("rval has %c ascii character value", rval);
            $display("pd strength value is %v", pd);
            $display("current scope is %m");
            $display("%s is ascii value for 101", 101);
```

```
            $write("simulation time is");
            $write("%t\n", $time);
        end
    endmodule
```

输出结果为

```
\%
"S
rval = 00000065 hex 101 decimal
rval = 00000000145 octal 00000000000000000000000001100101 binary
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is 0
```

在 $display 中，输出列表中数据的显示宽度是自动按照输出格式进行调整的。这样在显示输出数据时，在经过格式转换以后，总是用表达式的最大可能值所占的位数来显示表达式的当前值。

5.3.2　$monitor 和 $strobe

$monitor、$strobe 与 $display、$write 一样，属于信号的输出显示的系统任务。同时，$monitor 与 $strobe 都提供了监控和输出参数列表中字符或变量的值的功能。

(1) $monitor 的语法格式如下：

```
$monitor("<format_specifiers>", <signal1, signal2, …, signaln>);
```

例如：

```
$monitor(p1, p2, …, pn);
```

任务 $monitor 提供了监控和输出参数列表中的表达式或变量值的功能。其参数列表中输出控制格式字符串和输出列表的规则与 $display 中的一样。当启动一个带有一个或多个参数的 $monitor 任务时，仿真器则建立一个处理机制，使得每当参数列表中变量或表达式的值发生变化时，整个参数列表中变量或表达式的值都将输出显示。如果同一时刻两个或多个参数的值发生变化，则在该时刻只输出显示一次。

在 $monitor 中，参数可以是 $time 系统函数。这样，参数列表中变量或表达式的值同时发生变化的时刻可以通过标明同一时刻的多行输出来显示。例如：

```
$monitor($time, , "rxd = %b txd = %b", rxd, txd);
```

在 $display 中也可以这样使用。注意在上面的语句中，“, , ”代表一个空参数。空参数在输出时显示为空格。

$monitor 还提供了以下两种常用的系统任务：

```
$monitoron;
$monitoroff;
```

$monitoron 和 $monitoroff 任务的作用是通过打开和关闭监控标志来控制监控任务 $monitor 的启动和停止，这样，程序员可以很容易地控制 $monitor 的发生时间。其中 $monitoroff 任务用于关闭监控标志，停止监控任务 $monitor；$monitoron 则用于打开监控标志，启动监控任务$monitor。通常在通过调用 $monitoron 启动 $monitor 时，不管$monitor 参数列表中的值是否发生变化，总是立刻输出显示当前时刻参数列表中的值，这用于在监控的初始时刻设定初始比较值。在缺省情况下，控制标志在仿真的起始时刻就已经打开了。在多模块调试的情况下，许多模块中都调用了 $monitor，但因为任何时刻只能有一个 $monitor 起作用，因此需配合 $monitoron 与 $monitoroff 使用，把需要监视的模块用 $monitoron 打开，在监视完毕后及时用 $monitoroff 关闭，以便把 $monitor 让给其它模块使用。

$monitor 与 $display 的不同之处还在于 $monitor 往往在 initial 块中调用，只要不调用 $monitoroff，$monitor 便不间断地对所设定的信号进行监视。

例 5.3-2 $monitor 系统任务的应用示例。

```
module monitor_tb;
  integer a, b;
     initial begin
    a = 2;
    b = 4;
    forever
       begin
          #5 a = a + b;
          #5 b = a - 1;
       end
  end
  initial #40 $finish;
  initial
     begin
       $monitor($time, "a = %d, b = %d", a, b);
     end
endmodule
```

输出结果：

```
0    a =        2, b =      4
5    a =        6, b =      4
10   a =        6, b =      5
15   a =       11, b =      5
20   a =       11, b =     10
25   a =       21, b =     10
30   a =       21, b =     20
35   a =       41, b =     20
```

(2) $strobe 的语法格式如下：

```
$strobe(<functions_or_signals>);
$strobe("<string_and/or_variables>", <functions_or_signals>);
```

$strobe 系统任务用于某时刻所有时间处理完后，在这个时间步的结尾输出一行格式化的文本。Verilog HDL 提供了除$strobe 以外的其它几个相关的扩展系统任务，其含义如下：

$strobe——在所有时间处理完后，以十进制格式输出一行格式化的文本；

$strobeb——在所有时间处理完后，以二进制格式输出一行格式化的文本；

$strobeo——在所有时间处理完后，以八进制格式输出一行格式化的文本；

$strobeh——在所有时间处理完后，以十六进制格式输出一行格式化的文本。

这些系统任务在指定时间显示模拟数据，但这种任务的执行是在该特定时间步结束时才显示模拟数据。"时间步结束"意味着对于指定时间步内的所有时间都已经处理了。

$strobe 任务的参数定义和 $display 任务相同，但是 $strobe 任务在被调用时刻所有的赋值语句都完成后，才输出相应的文字信息。因此 $strobe 任务提供了另一种数据显示机制，可以保证数据只在所有赋值语句执行完毕后才被显示。

例 5.3-3 $strobe 系统任务的应用示例。

```
module strobe_tb;
    reg a, b;
        initial begin
          a = 0;
          $display("a by display is: ", a);
          $strobe("a by strobe is: ", a);
          a = 1;
    end
    initial begin
          b <= 0;
          $display("b by display is: ", b);
          $strobe ("b by strobe is: ", b);
          #5;
          $display("#5 b by display is: ", b);
          $strobe("#5 b by strobe is: ", b);
          b <= 1;
    end
endmodule
```

输出结果：

```
 a by display is:0
 b by display is:x
 a by strobe is:1
 b by strobe is:0
#5 b by display is:0
#5 b by strobe is:1
```

5.3.3 $time 和 $realtime

在 Verilog HDL 中有两种类型的时间系统函数：$time 和 $realtime。用这两个时间系统函数可以得到当前的仿真时刻。这两个函数被调用后都返回当前时刻相对仿真开始时刻的时间量值。所不同的是，$time 函数以 64 位整数值的形式返回仿真时间，而 $realtime 函数则以实型数据的形式返回仿真时间。

(1) 系统函数 $time。$time 可以返回一个 64 比特整数表示的当前仿真时刻值，该时刻值是以模块的开始仿真时间为基准的。

例 5.3-4 $time 系统任务的应用示例。

```
`timescale 1ns/1ns
module time_tb;
  reg ts;
  parameter delay = 2;
       initial begin
          #delay ts = 1;
          #delay ts = 0;
          #delay ts = 1;
          #delay ts = 0;
       end
       initial
          $monitor($time, , , "ts = %b", ts);     //使用函数$time
  endmodule
```

输出结果：

```
0    ts = x
2    ts = 1
4    ts = 0
6    ts = 1
8    ts = 0
```

(2) 系统函数 $realtime。$realtime 和 $time 的作用是一样的，只是 $realtime 返回的时间数字是一个实型数，该数字也是以时间尺度为基准的。

例 5.3-5 $realtime 系统任务的应用示例。

```
`timescale 1ns/1ns
module realtime_tb;
    reg set;
    parameter   p = 2;
      initial
           begin
             $monitor($realtime, , "set = b%", set);     //使用函数$realtime
             #p set = 0;
```

```
            #p set = 1;
        end
endmodule
```

输出结果：

```
0   set = x
2   set = 0
4   set = 1
```

5.3.4 $finish 和$stop

系统任务 $finish 和 $stop 用于对仿真过程的控制，分别表示结束仿真和中断仿真。其语法格式分别如下：

```
$finish;
$finish(n);
$stop;
$stop(n);
```

其中，n 是参数，可以取 0、1 或 2 几个值，含义如表 5.3-3 所示。

表 5.3-3　n 的取值及其含义

n 的取值	含　义
0	不输出任何信息
1	给出仿真时间和位置
2	给出仿真时间和位置，同时还有所用 memory 及 CPU 时间的统计

系统任务 $finish 的作用是退出仿真器，返回主操作系统，也就是结束仿真过程。任务 $finish 可以带参数，根据参数的值输出不同的特征信息；如果不带参数，默认 $finish 的参数值为 1。

系统任务 $stop 的作用是把 EDA 工具(例如仿真器)置成暂停模式，在仿真环境下给出一个交互式的命令提示符，将控制权交给用户。该任务可以带有参数表达式。根据参数值(0、1 或 2)的不同，输出不同的信息。参数值越大，输出的信息越多。

例 5.3-6　$finish 系统任务的应用示例。

```
module finish_tb;
  integer a, b;
  initial begin
    a = 2;
    b = 4;
    forever begin
      #5 a= a + b;
      #5 b= a - 1;
    end
  end
```

```
    initial #40 $finish;
    initial begin
      $monitor($time, "a = %d, b = %d", a, b);
    end
  endmodule
```

例中，程序执行到第 40 个时间单位时退出仿真器。

例 5.3-7　$stop 系统任务的应用示例。

```
module stop_tb;
  integer a, b;
  initial begin
    a = 2;
    b = 4;
    forever begin
      #5 a = a + b;
      #5 b = a - 1;
    end
  end
  initial #40 $stop;
  initial begin
    $monitor($time, "a = %d, b = %d", a, b);
  end
endmodule
```

例中，程序执行到第 40 个时间单位时停止仿真，将 EDA 仿真器设置为暂停模式。

5.3.5　$readmemh 和$readmemb

Verilog HDL 程序中的系统任务 $readmemb 和 $readmemh 用来从文件中读取数据到存储器中。这两个系统任务可以在仿真的任何时刻被执行使用，其语法格式分别如下：

```
$readmemb("<file_name>", <memory_name>);
$readmemb("<file_name >", <memory_name>, <start_addr>);
$readmemb("<file_name >", <memory_name>, <start_addr>, <finish_addr>);
$readmemh("<file_name >", <memory_name>);
$readmemh("<file_name >", <memory_name>, <start_addr>);
$readmemh("<file_name >", <memory_name>, <start_addr>, <finish_addr>);
```

在这两个系统任务中，被读取的数据文件的内容只能包含空白位置(空格、换行符、制表符(tab)和 form-feeds)、注释行(//形式的和/*…*/形式的都允许)、二进制或十六进制的数字。数字中不能包含位宽说明和格式说明，对于$readmemb 系统任务，每个数字必须是二进制数字；对于 $readmemh 系统任务，每个数字必须是十六进制数字。读取数据文件中的不定值(x 或 X)、高阻值(z 或 Z)和下划线(_)的使用方法及代表的意义与一般 Verilog HDL 程序中的使用方法及代表的意义是一样的。另外，数字必须用空白位置或注释行分隔开。

例 5.3-8　$readmemh 和 $readmemb 系统任务的应用示例。

```
module read_mem_tb;
  reg [7:0] memory_b [0:7];
  reg [31:0] memory_h [0:31];
  integer i;
  initial begin
    //把数据文件 init_b.txt 读入存储器中的给定地址
    $readmemb("init_b.txt", memory_b);
    //把数据文件 init_h.txt 读入存储器中的给定地址
    $readmemh("init_h.txt", memory_h);
    //显示初始化后的存储器内容
    for(i = 0; i<8; i = i+1)
    begin
      $display("memory_b [%0d] = %b", i, memory_b[i]);
      $display("memory_h [%0h] = %h", i, memory_h[i]);
    end
  end
endmodule
```

文件 init_b.txt 和 init_h.txt 包含初始化数据。用@<address>在数据文件中指定地址。其中，“init_b.txt”指定二进制数据从第二位地址开始写入；而“init_h.txt”指定十六进制数据从第一位地址写入。样本文件如下。

init_b.txt 文件：

```
@002
11111111 01010101
00000000 10101010
@006
1111zzzz 00001111
```

init_h.txt 文件：

```
@001
00000000000000000000000000000011
00000000000000000000000000000111
00000000000000000000000000001111
00000000000000000000000000011111
```

5.3.6　$random

$random 是产生随机数的系统函数，每次调用该函数将返回一个 32 位的随机数，该随机数是一个带符号的整型数。其语法格式如下：

```
$random%<number>;
```

该系统函数提供了一个产生随机数的手段。当函数被调用时返回一个 32 bit 的随机数。

它是一个带符号的整型数。

$random 的一般用法如下：

```
$random % b;
```

其中，b 为一常数且要求大于零，它给出了一个范围在 −b+1 到 b−1 之间的随机数。

利用位拼接操作符“{ }”将函数 $random 返回的有符号整数变换为无符号数，其用法如下：

```
{$random}%b;
```

其中，b 为一常数且要求大于零，它给出了一个范围在 0 到 b−1 之间的随机数。

例 5.3-9　$random 系统任务的应用示例。

```
`timescale 1ns/1ns
module random_pulse(dout);
    output [9:0] dout;
    reg dout;
    integer delay1, delay2, k;
      initial
         begin
         #10 dout = 0;
         for (k = 0; k < 100; k = k+1)
           begin
             delay1 = 20 * ( {$random} % 6);              //delay1 在 0～100 ns 的范围变化
             delay2 = 20 * ( 1 + {$random} % 3);          //delay2 在 20～60 ns 的范围变化
             #delay1   dout = 1 << ({$random} %10);
             //dout 的 0～9 位中随机出现 1，且出现的时间在 0～100 ns 的范围变化
             #delay2   dout = 0;
             //脉冲的宽度在 20～60 ns 的范围变化
           end
      end
endmodule
```

5.3.7　值变转储文件系统任务

值变转储 VCD(Value Change Dump)文件是一个 ASCII 文件，它包括仿真时间、范围与信号的定义以及仿真运行过程中信号值的变化等信息。设计中的所有信号或者选定的信号集合在仿真过程中都可以被写入 VCD 文件。后处理工具可以把 VCD 文件作为输入，并把层次信息、信号值和信号波形显示出来。对于大规模设计的仿真，设计者都可以把选定的信号转储到 VCD 文件中，并使用后处理工具调试、分析和验证仿真输出结果。调试过程中 VCD 文件的使用流程如图 5.3-1 所示。

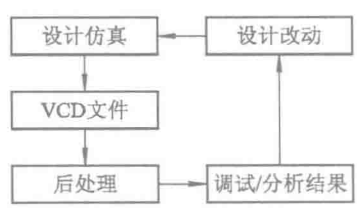


图 5.3-1　调试过程中 VCD 文件的使用流程

Verilog HDL 提供了系统任务来选择要转储的模块实例或者模块实例信号($dumpvars)，选择 VCD 文件的名称($dumpfile)，选择转储过程的起点和终点($dumpon 和$dumpoff)，选择生成监测点($dumpall)。

(1) $dumpfile 用于指定 VCD 文件名。若不指定 VCD 文件，则由仿真器指定一个默认文件。例如，仿真信息转储到 myfile.dmp 文件：

```
$dumpfile("myfile.dmp");
```

(2) $dumpvars 用于转储模块中的信号。

例如，若没有指定变量范围，则把设计中的全部信号都转储：

```
$dumpvars;
```

转储模块实例 top 中的信号：

```
$dumpvars(1, top);          //只转储 top 下的第一层信号
$dumpvars(2, top.m1);       //转储 top.m1 模块以下两层信号
$dumpvars(0, top.m1);       //数字 0 表示转储 top.m1 模块下面各层的所有信号
```

(3) $dumpon 和 $dumpoff 用于启动和停止转储过程。例如：

```
initial
  begin
        $dumpon;            //启动转储过程
        #1000 $dumpoff;     //过了 1000 个仿真时间单位后，停止转储过程
  end
```

(4) $dumpall 用于生成一个检查点，转储所有 VCD 变量的当前值。

(5) $dumplimit 用于指定 VCD 文件的最大长度(字节)。例如：

```
$dumplimit(1024);           //VCD 文件的最大长度为 1024 字节
```

(6) $dumpflush 用于刷新操作系统 VCD 文件缓冲区中的数据。例如：

```
$dumpflush;                 //转储恢复到刷新以前的状态
```

$dumpfile 和 $dumpvars 任务通常在仿真开始时指定。$dumpon、$dumpoff 和 $dumpall 任务在仿真过程中控制转储过程。表 5.3-4 是扩展的 VCD 文件。

表 5.3-4　扩展的 VCD 文件

名　称	含　义	举　例
$dumpports	用于将所列出的模块端口的值转储到指定的文件中	$dumpports(uart, "dport.dump");
$dumpportsall	用于转储所有被选择的端口值，即使它们的值可能没有发生变化	$dumpportsall("dport.dump");
$dumpportsoff	用于关闭端口值的转储	$dumpportsoff("dport.dump");
$dumpportson	用于打开端口值的转储	$dumpportson("dport.dump");
$dumpportslimit	用于限制 VCD 文件的长度	$dumpportslimit(1024, "dport.dump");
$dumpportsflush	用于把系统内部缓存的数据转储到文件中	$dumpportsflush("dport.dump");

例 5.3-10 以脉冲计数器为例说明 VCD 的使用。

程序如下：

```
module count(clk, count);
  input clk;
  output reg[0:3] count;
  initial      count = 'd1;
  always@(posedge clk)
      begin
          count = count + 1;
      end
endmodule
```

测试程序如下：

```
module counter_tb;
  reg clock, up_dn;
  wire [0:3] cnt_out;
  parameter ON_DELAY = 1, OFF_DELAY = 2;
  localparam DUMP_FILE ="count.vcd";
  localparam DUMP_DONE = 200;
  count ucud(clock, cnt_out);
  always
    begin
      clock = 1;
      #ON_DELAY;
      clock = 0;
      #OFF_DELAY;
    end
  initial
    begin
      #DUMP_DONE $dumpflush;
      $stop;
    end
  initial
    begin
      $dumpfile(DUMP_FILE);
      $dumplimit(1024);
      $dumpvars(0, test_counter);
      $dumpvars(0, ucud.count, ucud.clk, );
    end
  initial
```

```
    begin
      $dumpportslimit(10000, "count.evcd");
      #10;
      $dumpports(ucud, "count.evcd");
      #40;
      $dumpportsall("count.evcd");
      #30;
      $dumpportsoff("count.evcd");
      #20;
      $dumpportson("count.evcd");
      $dumpportsall("count.evcd");
      $dumpportsflush("count.evcd");
    end
endmodule
```

VCD 文件是 ASCII 文件，所包含的信息如下：

(1) 文件头信息：提供日期、仿真器版本和时间刻度单位。

(2) 节点信息：定义所转储变量的范围和类型。

(3) 值的变化情况：实际取值随时间变化的信息，同时记录绝对仿真时间。

其中生成的拓展 VCD 文件如下：

```
$date
    Fri Apr 08 14:59:53 2011
$end
$version
    ModelSim Version 6.5c
$end
$timescale
    1ns
$end
$scope module counter_tb $end
$var parameter 32 ! ON_DELAY $end
$var parameter 32 " OFF_DELAY $end
$var parameter 72 # DUMP_FILE $end
$var parameter 32 $ DUMP_DONE $end
$var reg 1 % clock $end
$var reg 1 & up_dn $end
$var wire 1 ' cnt_out [0] $end
$var wire 1 ( cnt_out [1] $end
$var wire 1 ) cnt_out [2] $end
$var wire 1 * cnt_out [3] $end
```

```
#0
$dumpvars
b1 !
b10 "
b110001101101111011101010110111001
11010000101110011101100110001101100100
#
...
...
#22
0%
0+
#24
1%
1+
b1010 ,
0*
1)
#25
0%
```

```
$scope module ucud $end
$var wire 1 + clk $end
$var reg 4 , count [0:3] $end
$var wire 1 + clk $end
$upscope $end
$upscope $end
$enddefinitions $end
0+
#27
1%
1+
b1011 ,
1*
$comment dumplimit size reached $end
```

有一些具有图形显示功能的后处理工具可以供商业上使用，它们目前是仿真和调试过程的重要组成部分。对于大规模的仿真，设计者难以分析 $display 和$monitor 语句的输出。从仿真波形图来分析结果更加直观。VCD 之外的其它格式也已经出现，但是 VCD 仍然是最流行的 Verilog HDL 仿真器转储格式。

VCD 文件可能变得非常庞大(对大规模设计而言，VCD 文件的大小可能达到数百兆字节)，因而只能选择那些需要检查的信号进行转储。

VCD 数据库是仿真过程中数据信号变化的记录，它只记录用户指定的信号。使用 VCD 系统任务时需要注意如下事项：

(1) 用户可以用 $dump* 系统任务打开一个数据库，保存信号并控制信号的保存。除 $dumpvars 外，其它任务的作用都比较直观。

(2) 必须首先使用 $dumpfile 系统任务，并且在一次仿真中只能打开一个 VCD 数据库。

(3) 在仿真前(时间 0 前)必须先指定要观测的波形，这样才能看到信号完整的变化过程。

(4) 在仿真时，定期地将数据保存到磁盘是一个良好的习惯，因为一旦系统出现问题数据也不会全部丢失。

(5) VCD 数据库不记录仿真结束时的数据。因此如果希望看到最后一次数据变化后的波形，必须在结束仿真前使用 $dumpall。

5.4　信号时间赋值语句

在集成电路设计和验证阶段，经常需要对特定信号进行延时来实现相应的时序控制，或者避免信号冲突形成电路中的热点。信号的时间延迟可以通过两类方式来完成：一类是延时控制，它是为行为语句的执行指定一个延迟时间的信号时间延迟方式，可以分为串行延时控制、并行延时控制、阻塞式延时控制和非阻塞式延时控制；另一类是事件控制，它是为行为语句的执行指定触发事件的信号时间延迟方式，可以分为边沿触发事件控制和电平敏感事件控制。图 5.4-1 为信号的时间延迟分类。



图 5.4-1　信号的时间延迟分类

5.4.1 时间延迟的语法说明

延时语句用于对各条语句的执行时间进行控制，从而快速满足用户的时序要求。Verilog HDL 中延时控制的语法格式有如下两类：

```
#<延迟时间>行为语句;
```

和

```
#<延迟时间>;
```

其中，符号“#”是延时控制的关键字符，“延迟时间”可以是直接指定的延迟时间量，或者以参数的形式给出。在仿真过程中，所有延迟时间都根据时间单位定义。例如：

```
#2 Sum = A ^ B;    //#2 指定 2 个时间单位后，将 A 异或 B 的值赋给 Sum
```

根据时间控制部分在过程赋值语句中出现的位置，可以把过程赋值语句中的时间控制方式分为外部时间控制方式和内部时间控制方式。

(1) 外部时间控制方式：时间控制出现在整个过程赋值语句的最左端，也就是出现在赋值目标变量的左边的时间控制方式。

其语法结构举例如下：

```
#5 a = b;
```

这在仿真执行时就相当于执行了如下几条语句：

```
initial
  begin
     #5;
     a = b;
  end
```

(2) 内部时间控制方式：过程赋值语句中的时间控制部分还可以出现在“赋值操作符”和“赋值表达式”之间的时间控制方式。

其语法结构举例如下：

```
a = #5b;
```

其中，时间控制部分“#5”就出现在赋值操作符“=”和赋值表达式“b”的中间，因此在这条过程赋值语句内带有内部时间控制方式的时间控制。它在执行时就相当于执行了如下几条语句：

```
initial
  begin
    temp = b;   //先求 b 的值
    #5;
    a = temp;
  end
```

5.4.2 时间延迟的描述形式

此处，时间延迟的描述形式是指延时控制的描述形式，分为串行延时控制、并行延时控制、阻塞式延时控制和非阻塞式延时控制四种形式。下面以实现两组不同波形的信号为

例(图 5.4-2 中的 q0_out 和 q1_out)来说明四种不同时间延迟的描述形式。

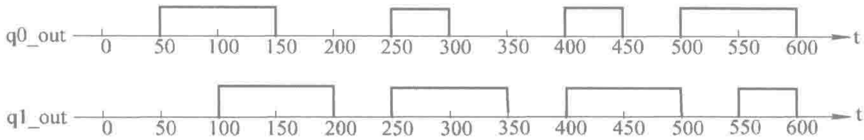


图 5.4-2 两组示例波形

1. 串行延时控制

串行延时控制是最为常见的信号延时控制，它是由 begin-end 过程块加上延时赋值语句构成，其中延时赋值语句可以为外部时间控制方式，也可以为内部时间控制方式。在“延迟时间”之后也可根据实际情况来确定是否执行相应的行为语句。

若在“延迟时间”后面有相应的行为语句，则仿真进程遇到这条带有延时控制的行为语句后并不立即执行行为语句指定的操作，而是等到“延迟时间”所指定的时间量过去后才真正开始执行行为语句指定的操作。

例 5.4-1 以 Verilog HDL 串行延时控制方式设计图 5.4-2 所示信号。

```
`timescale 1ns/1ns
module serial_delay (q0_out, q1_out);
  output q0_out, q1_out;
  reg   q0_out, q1_out;
  initial
    begin
             q0_out = 1'b0;
        #50   q0_out = 1'b1;
        #100  q0_out = 1'b0;
        #100  q0_out = 1'b1;
        #50   q0_out = 1'b0;
        #100  q0_out = 1'b1;
        #50   q0_out = 1'b0;
        #50   q0_out = 1'b1;
        #50   q0_out = 1'b0;
    end
  initial
    begin
             q1_out = 1'b0;
        #100  q1_out = 1'b1;
        #100  q1_out = 1'b0;
        #50   q1_out = 1'b1;
        #100  q1_out = 1'b0;
        #50   q1_out = 1'b1;
        #100  q1_out = 1'b0;
```

```
            #50   q1_out = 1'b1;
            #50   q1_out = 1'b0;
      end
  endmodule
```

在上述代码中，产生指定的输出信号 q0_out 和 q1_out 是通过串行延时控制方式实现的，即每一个延时赋值语句是在其前一个赋值语句执行完成后，延迟相应的时间单位，才开始执行当前的赋值语句。

2. 并行延时控制

并行延时控制方式是通过 fork-join 过程块加上延时赋值语句构成的，其中延时赋值语句同串行延时控制方式一样，既可以是外部时间控制方式，也可以是内部时间控制方式。在“延迟时间”之后也可根据实际情况来确定是否执行相应的行为语句。

在“延迟时间”后面有相应的行为语句，则仿真进程遇到这条带有延时控制的行为语句后并不立即执行行为语句指定的操作，而是等到“延迟时间”所指定的时间量过去后才真正开始执行行为语句指定的操作。但并行延时控制方式与串行延时控制方式的不同之处在于，并行延时控制方式中的多条延时语句是并行执行的，并不需要等待上一条语句执行完成后才开始执行当前的语句。

例 5.4-2 以 Verilog HDL 并行延时控制方式设计图 5.4-2 所示信号。

```
`timescale 1ns/1ns
module parallel_delay(q0_out, q1_out);
  output q0_out, q1_out;
  reg   q0_out, q1_out;
  initial
     fork
           q0_out = 1'b0;
     #50   q0_out = 1'b1;
     #150q0_out = 1'b0;
     #250q0_out = 1'b1;
     #300q0_out = 1'b0;
     #400q0_out = 1'b1;
     #450q0_out = 1'b0;
     #500q0_out = 1'b1;
     #600q0_out = 1'b0;
     join
  initial
     fork
           q1_out = 1'b0;
     #100q1_out = 1'b1;
     #200q1_out = 1'b0;
     #250q1_out = 1'b1;
```

```
        #350q1_out = 1'b0;
        #400q1_out = 1'b1;
        #500q1_out = 1'b0;
        #550q1_out = 1'b1;
        #600q1_out = 1'b0;
        join
    endmodule
```

在上述代码中，产生指定的输出信号 q0_out 和 q1_out 是通过并行延时控制方式实现的，即每一个延时赋值语句不用等待前一个延时赋值语句执行完成后才开始执行，所有的延时赋值语句都是在仿真进程的起始时刻开始执行的。

3. 阻塞式延时控制

以赋值操作符“=”来标识的赋值操作称为“阻塞式过程赋值”，阻塞式过程赋值之前已经介绍过，在此介绍阻塞式延时控制。阻塞式延时控制是在阻塞式过程赋值基础上带有延时控制的情况。例如：

```
initial
  begin
    a = 0;
    a = #5    1;
    a = #10   0;
    a = #15   1;
  end
```

各条阻塞式赋值语句将依次得到执行，并且在第一条语句所指定的赋值操作没有完成之前第二条语句不会开始执行。因此在仿真进程开始时刻将“0”值赋给 a，此条赋值语句完成之后才开始执行第二条赋值语句；在完成第一条赋值语句之后，延迟 5 个时间单位将“1”值赋给 a；同理第三条赋值语句是在第二条赋值语句完成之后延迟10个时间单位才开始执行，将“0”值赋给 a；最后一条赋值语句是在前三条语句都完成后，延迟 15 个时间单位，将“1”值赋给 a。图 5.4-3 给出了该例中信号 a 的波形。例 5.4-1 和例 5.4-2 都采用的是阻塞式延时控制方式。

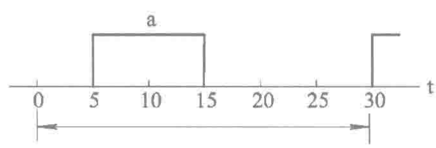


图 5.4-3 阻塞式延时控制

4. 非阻塞式延时控制

以赋值操作符“<=”来标识的赋值操作称为“非阻塞式过程赋值”，非阻塞式过程赋值也在之前介绍过，在此主要介绍非阻塞式延时控制。非阻塞式延时控制是在非阻塞式过程赋值基础上带有延时控制的情况。例如：

```
initial
  begin
        a <=      0;
        a <= #5   1;
        a <= #10  0;
```

```
        a <= #15  1;
    end
```

例中，各条非阻塞式赋值语句均以并行方式执行，虽然执行语句在 begin-end 串行块中，但其执行方式与并行延时控制方式一致，在仿真进程开始时刻同时执行四条延时赋值语句。在仿真进程开始时，将“0”值赋给 a；在距仿真开始时刻 5 个时间单位时，将“1”值赋给 a；在距仿真开始时刻 10 个时间单位时，将“0”值赋给 a；最后，在距仿真开始时刻 15 个时间单位时，将“1”值赋给 a。图 5.4-4 给出了该例中信号 a 的波形。

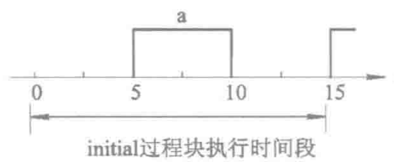


图 5.4-4　非阻塞式延时控制

例 5.4-3　以 Verilog HDL 非阻塞延时控制方式设计图 5.4-2 所示信号。

```
`timescale 1ns/1ns
module non_blocking_delay(q0_out, q1_out);
  output q0_out, q1_out;
  reg   q0_out, q1_out;
  initial
     begin
          q0_out <=       1'b0;
          q0_out <= #50  1'b1;
          q0_out <= #150 1'b0;
          q0_out <= #250 1'b1;
          q0_out <= #300 1'b0;
          q0_out <= #400 1'b1;
          q0_out <= #450 1'b0;
          q0_out <= #500 1'b1;
          q0_out <= #600 1'b0;
     end
  initial
     begin
          q1_out <=       1'b0;
          q1_out <= #100 1'b1;
          q1_out <= #200 1'b0;
          q1_out <= #250 1'b1;
          q1_out <= #350 1'b0;
          q1_out <= #400 1'b1;
          q1_out <= #500 1'b0;
          q1_out <= #550 1'b1;
          q1_out <= #600 1'b0;
     end
```

```
endmodule
```

在上述代码中，产生指定的输出信号 q0_out 和 q1_out 是通过非阻塞式延时控制方式实现的，延迟时间采用的是内部时间控制方式，赋值语句的执行方式是并行的，每一条延时赋值语句的执行不需要等待上一条延时赋值语句执行完成之后才执行。其执行过程与并行延时控制方式是一致的。

5.4.3 边沿触发事件控制

边沿触发事件控制方式是在指定的信号变化时刻，即指定的信号跳变边沿才触发语句的执行，而当信号处于稳定状态时则不会触发语句的执行。

边沿触发事件控制的语法格式有如下四种形式：

```
@(<事件表达式>)行为语句;
@(<事件表达式>);
@(<事件表达式 1>or<事件表达式 2>or…or<事件表达式 n>)行为语句;
@(<事件表达式 1>or<事件表达式 2>or…or<事件表达式 n>);
```

在以上四种格式中，符号“@”是边沿触发事件控制的标识符；“事件表达式”代表着触发语句执行的触发事件；“行为语句”则指出了触发事件所要触发执行的具体操作。

格式中的事件表达式部分，即(<事件表达式>)和(<事件表达式 1>or<事件表达式 2>or…or<事件表达式 n>)被称为“敏感事件列表”。

1. 事件表达式

事件表达式中可以出现以下三种形式：

形式 1：

<信号名>

形式 2：

posedge<信号名>

形式 3：

negedge<信号名>

其中，“信号名”可以是任何数据类型的标量或矢量。

说明：

(1) 形式 1 中，代表触发事件的“信号名”在指定的信号发生逻辑变化时，执行下面的语句(示例)：

```
@(in) out = in;
```

当敏感事件 in 发生逻辑变化(包括正跳变和负跳变)时，执行对应的赋值语句，将 in 的值赋给 out。

(2) 形式 2 中，代表触发事件的“posedge<信号名>”在指定的信号发生了正跳变时，执行下面的语句(示例)：

```
@(posedge in) out = in;
```

当敏感事件 in 发生正跳变时，执行对应的赋值语句，将 in 的值赋给 out。

(3) 形式 3 中，代表触发事件的“negedge<信号名>”在指定的信号发生了负跳变时，执行下面的语句(示例)：

```
@(negedge in)out = in;
```

当敏感事件 in 发生负跳变时，执行对应的赋值语句，将 in 的值赋给 out。

在信号发生逻辑变化(正跳变或负跳变)的过程中，信号的值是从 0、1、x、z 四个值中的一个值变化到另一个值；而信号发生正跳变的过程是信号由低电平向高电平的转变，负跳变是信号由高电平向低电平的转变。表 5.4-1 为 Verilog HDL 中规定的逻辑信号的正跳变和负跳变。

表 5.4-1　逻辑信号的正跳变和负跳变

正 跳 变	负 跳 变
0→x	1→x
0→z	1→z
0→1	1→0
x→1	x→0
z→1	z→0

2. 边沿触发语法格式

形式 1：

```
@(<事件表达式>)行为语句;
```

这种语法格式的“敏感事件列表”内只包含了一个触发事件，只有当这个指定的触发事件发生之后，后面的行为语句才能启动执行。在仿真进程中遇到这种带有事件控制的行为语句时，如果指定的触发事件还没有发生，则仿真进程就会停留在此处等待，直到指定触发事件发生后才启动执行后面的行为语句，仿真进程继续向下进行。

例 5.4-4　时钟脉冲计数器示例。

```
module clk_counter(clk, count_out);
  input clk;
  output count_out;
  reg [3:0] count_out;
  initial
    count_out = 0;
  always@(posedge clk)
    count_out = count_out + 1;      //在 clk 的每个正跳变边沿 count_out 增加 1
endmodule
```

在事件控制语句中，触发事件时钟信号“clk”发生正跳变时，计数寄存器“count_out”增加 1。

形式 2：

```
@(<事件表达式>);
```

这种语法格式的“敏感事件列表”内也只包含了一个触发事件，没有行为语句来指定触发事件发生时要执行的操作。其事件控制语句的执行过程与延时控制语句中没有行为语句的情况类似，仿真进程在遇到这条事件控制语句后会进入等待状态，直到指定的触发事件发生后才结束等待状态，退出该事件控制语句的执行并开始下一条语句的执行。

例 5.4-5　用于测定输入时钟正电平、负电平持续时间以及时钟周期的模块示例。

```
module clk_time_mea(clk);
  input clk;
  time posedge_time, negedge_time;
  time high_last_time, low_last_time, last_time;
  initial
    begin
      @(posedge clk);            /*等待，直到时钟发生正跳变后退出等待状态，
                                   继续执行下一条语句*/
      posedge_time = $time;
      @(negedge clk);            /*等待，直到时钟发生负跳变后退出等待状态，
                                   继续执行下一条语句*/
      negedge_time = $time;
      @(posedge clk);            /*等待，直到时钟再次正跳变后退出等待状态，
                                   继续执行下一条语句*/
      last_time = $time - posedge_time;
      high_last_time = negedge_time - posedge_time;
      low_last_time = last_time - high_last_time;
      $display("The clk stay in High level for:%t", high_last_time);
      $display("The clk stay in Low level for:%t", low_last_time);
      $display("The clk signal Period is:%t", last_time);
    end
endmodule
```

例中，三条边沿触发事件控制语句的作用是仿真进程在遇到这几条语句时进入等待状态，分别等待时钟信号正跳变、负跳变和再次正跳变的发生。当发生了对应的跳变之后仿真进程就退出等待状态，结束对应事件控制语句的执行并开始下一条语句的执行，保存当前的仿真时间值。

形式 3：

```
@(<事件表达式 1>or<事件表达式 2>or…or<事件表达式 n>)行为语句;
```

这种语法格式的“敏感事件列表”内指定了由不同“事件表达式”代表的多个触发事件，这些“事件表达式”之间要用关键词“or”组合起来。只要这些触发事件中的任何一个发生，就启动行为语句的执行。在仿真进程遇到这种格式的边沿触发事件控制语句时如果所有的触发事件都没有发生，则仿真进程就会进入等待状态，直到其中的某一个触发事件发生后才启动执行后面给出的行为语句，仿真进程继续向下进行。

形式 4：

```
@(<事件表达式 1>or<事件表达式 2>or…or<事件表达式 n>);
```

同第三种语法格式一样，这种语法格式内指定了多个触发事件。但是在这种格式中没有行为语句。在这种情况下，该语句的执行过程与第二种语法格式的执行过程类似，仿真进程在遇到这条事件控制语句后会进入等待状态，直到敏感事件列表包含的多个触发事件中的任何一个发生后才结束等待状态，退出该事件控制语句并开始执行该事件控制语句后

的下一条语句。

例 5.4-6　在触发事件发生后退出事件控制语句示例。

```
module display_information_change(a, b);
  input a, b;
  wire a, b;
  always
    begin
      @(posedge a or negedge b);          /*等待，直到 a 或 b 发生变化后退出等待状态，
                                            并开始下一条语句的执行*/
      display("One of a and b changed in time:%t", $time);
    end
endmodule
```

例中的模块在仿真时，仿真进程遇到事件控制语句后就进入等待状态，直到信号 a 正跳变或者信号 b 负跳变时才退出等待状态，结束事件控制语句的执行，开始下面语句的执行，显示当前仿真时刻的时间值(也就是 a 或 b 发生跳变时的时间值)。

5.4.4　电平敏感事件控制

电平敏感事件控制是另一种事件控制方式，与边沿触发事件控制不同，它是在指定的条件表达式为真时启动需要执行的语句。电平敏感事件控制是用关键词 wait 来表示的。

电平敏感事件控制的语法格式有如下两种：

形式 1：

```
wait(条件表达式)行为语句;
```

形式 2：

```
wait(条件表达式);
```

说明：

(1) 电平敏感事件控制的第 1 种形式中包含了“行为语句”，它可以是串行块(begin-end)语句或并行块(fork-join)语句，也可以是单条行为语句。在这种事件控制语句形式下，行为语句启动执行的触发条件是条件表达式的值为“真(逻辑 1)”。如果当仿真进程执行到这条电平敏感事件控制语句时条件表达式的值是“真”，那么语句块立即得到执行，否则语句块要一直等到条件表达式的值变为“真”时再开始执行。例如：

```
wait(enable == 1)
  begin
    d = a & b;
    d = d | c;
  end
```

wait 语句的作用是根据条件表达式的真假来控制其后 begin-end 语句块的执行。在使能信号 enable 变为高电平后，也就是 enable==1 的语句为真时进行 a、b、c 之间的与或逻辑操作；若使能信号 enable 未变为高电平，则 begin-end 语句块的执行需要等到 enable 变为高电平之后才开始执行。

(2) 电平敏感事件控制的第 2 种形式中没有包含“行为语句”。在这种电平敏感事件控制语句形式下，如果当仿真进程执行到该 wait 控制语句时条件表达式的值是“真”，那么立即结束该 wait 事件控制语句的执行，仿真进程继续往下进行；而如果当仿真进程执行到这条 wait 控制语句时条件表达式的值是“假”，则仿真进程进入等待状态，一直等到条件表达式取值变为“真”时才退出等待状态，同时结束该 wait 语句的执行，仿真进程继续向下进行。这种形式的电平敏感事件控制常常用来对串行块中各条语句的执行时序进行控制。例如：

```
begin
  wait(enable == 1);
  d = a & b;
  d = d | c;
end
```

该语句也能实现“等待使能信号 enable 取值变为 1 后进行 a、b、c 之间的与或逻辑操作”的功能。在仿真进程中，先执行不带行为语句的 wait 语句，这条语句的功能是等待使能信号 enable 变为 1 时，执行下面的与逻辑和或逻辑运算；若使能信号未变为 1，则仿真进程一直处于等待状态，等待使能信号 enable 变为 1。需要注意的是，wait 语句只有在串行结构的语句块中才能实现时序控制功能。

5.5 任务和函数

在行为级设计中，设计者经常需要在程序的多个不同地方实现同样的功能。因此有必要把这些公共部分提取出来组成子程序，然后在需要的地方调用这些子程序，这样就可以避免重复编码。在 Verilog HDL 中提供了任务和函数，可以将较大的行为级设计划分为较小的代码段，允许设计者将需要在多个地方重复使用的相同代码提取出来，编写成任务和函数，这样可以使代码更加简洁和易懂。

任务具有输入、输出和输入/输出(双向)变量，而函数具有输入变量，这样数据能够传入任务和函数，并且能够将结果输出。

5.5.1 任务

Verilog HDL 使用关键字 task 和 endtask 对任务进行声明。Verilog HDL 中规定，如果子程序满足以下任一条件，则公共子程序的描述必须使用任务而不能使用函数：

(1) 子程序中包含有延时、时序或者事件控制结构。

(2) 没有输出或者输出变量的数量大于 1。

(3) 没有输入变量。

任务(task)类似于其它高级程序编程语言中的“过程”。任务的使用包括任务的定义和任务的调用。

1. 任务的定义

任务定义的语法格式如下：

```
task<任务名>;
```

```
    端口和类型声明
    局部变量声明
    begin
        语句 1;
        语句 2;
        …
        语句 n;
    end
endtask
```

说明：

(1) 任务定义是嵌入在关键字 task 和 endtask 之间的，其中关键词 task 标志着一个任务定义结构的开端，endtask 标志着一个任务定义结构的结束。“任务名”是所定义任务的名称。在“任务名”后面不能出现输入、输出端口列表。

(2) “端口和类型声明”用于对任务各个端口的宽度和类型进行声明，其中使用关键词 input、output 或 inout 对任务的端口进行声明。input 类型和 inout 类型的变量从外部传递到任务中，input 类型的变量在任务所包含的语句中进行处理。当任务完成时，output 类型和 inout 类型的变量传回给任务调用语句中相应的变量。

在任务中使用关键字 input、output 或 inout 对输入/输出变量进行声明，其声明语句的语法与模块中端口声明的语法一样，但是两者在本质上是有区别的：模块的端口用来和外部信号相连接，而任务的输入/输出变量则用来向任务中传入或从中传出变量。

(3) “局部变量声明”用来对任务内用到的局部变量进行宽度和类型声明，该声明语句的语法与进行模块定义时的相应声明语句语法一致。

(4) 任务中由 begin 与 end 关键词界定的一系列语句指明了任务被调用时需要进行的操作，在任务被调用时这些语句将以串行方式得到执行。

例 5.5-1 以读存储器数据为例说明任务定义的操作。

```
task read_memory;               //任务定义的开头，指定任务名为 read_memory
    input[15:0]address;         //输入端口声明
    output[31:0]data;           //输出端口声明
    reg[3:0]counter;            //变量类型声明
    reg[7:0]temp[1:4];          //变量类型声明
    begin                       //语句块，任务被调用时执行
        for(counter = 1; counter <= 4; counter = counter+1)
        temp[counter] = mem[address+counter-1];
        data = {temp[1], temp[2], temp[3], temp[4]};
    end
endtask                         //任务定义结束
```

该例定义了一个名为 read_memory 的任务，用来读取存储器 mem 中的数据，该任务由一个 16 位的输入端口 address、一个 32 位的输出端口 data、一个 4 位的局部变量 counter 和一个 8 位的存储器 temp 构成。当该例所定义的任务被调用时，begin 和 end 之间的语句得到

执行，它们用来执行对存储器 mem 进行的四次读操作，将其结果合并后输出到 32 位的输出端口 data。

任务定义时的注意事项：

(1) 在第一行 task 语句中不能列出端口名列表。

(2) 任务中可以有延时语句、敏感事件控制语句等事件控制语句。

(3) 任务可以没有或可以有一个或多个输入、输出和双向端口。

(4) 任务可以没有返回值，也可以通过输出端口或双向端口返回一个或多个值。

(5) 任务可以调用其它的任务或函数，也可以调用该任务本身。

(6) 任务定义结构内不允许出现过程块(initial 或 always 过程块)。

(7) 任务定义结构内可以出现 disable 终止语句，这条语句的执行将中断正在执行的任务。在任务被中断后，程序流程将返回到调用任务的地方继续向下执行。

2. 任务的调用

任务的调用是通过“任务调用语句”来实现的。任务调用语句列出了传入任务和传出任务的参数值，任务的调用格式如下：

<任务名>(端口 1，端口 2，…，端口 n)；

在任务调用时任务调用语句只能出现在过程块内，任务调用语句就像一条普通的行为语句得到处理。当被调用的任务具有输入或输出端口时，任务调用语句必须包含端口名列表，这个列表内各个端口名出现的顺序和类型必须与任务定义结构中端口说明部分的端口顺序和类型一致。注意只有寄存器类的变量才能与任务的输出端口相对应。

例 5.5-2 以测试仿真中常用的方式来说明任务的调用。

```
module demo_task_invo_tb;
  reg [7:0] mem [127:0];
  reg [15:0] a;
  reg [31:0] b;
  initial
  begin
    a = 0;
    read_mem(a, b);            //任务的第一次调用
    #10;
    a = 64;
    read_mem(a, b);            //任务的第二次调用
  end
  task read_mem;               //任务定义部分
    input [15:0] address;
    output [31:0] data;
    reg [3:0] counter;
    reg [7:0] temp [1:4];
    begin
      for(counter = 1; counter <= 4; counter = counter+1)
```

```
            temp[counter] = mem[address+counter-1];
            data = {temp[1], temp[2], temp[3], temp[4]};
        end
    endtask
endmodule
```

在上面的模块中，任务 read_mem 调用了两次，由于这个任务在定义时说明了输入端口和输出端口，所以任务调用语句内必须包含端口名列表(a，b)，其中变量 a 与任务的输入端口 address 相对应，变量 b 与任务的输出端口 data 相对应，并且这两个变量在宽度上也是与对应的端口相一致的。这样，在任务被调用执行时，变量 a 的值通过输入端口传给了 address；在任务调用完成后，输出信号 data 又通过对应的端口传给了变量 b。

下面以实际中的交通灯控制为例说明任务定义和调用的特点。

例 5.5-3 用 task 描述交通灯的产生。

```
module traffic_lights(red, amber, green);
  output red, amber, green;
  reg [2:1] order;
  reg clock, red, amber, green;
  parameter ON = 1, OFF = 0, RED_TICS = 350, AMBER_TICS = 30, GREEN_TICS = 200;
  //产生时钟脉冲
  always
      begin
        #100 clock = 0;
        #100 clock = 1;
      end
  //任务的定义，该任务用于实现交通灯的开启
  task light;
    output red;
    output amber;
    output green;
    input [31:0] tic_time;
    input [2:1] order;
    begin
      red = OFF;
      green = OFF;
      amber = OFF;
      case(order)
        2'b01: red = ON;
        2'b10: green = ON;
        2'b11: amber = ON;
        default:{red, green, amber} = {ON, ON, ON};
```

```
            endcase
            repeat(tic_time)@(posedge clock);
            red = OFF;
            green = OFF;
            amber = OFF;
          end
        endtask
    //任务的调用，交通灯初始化
        initial
          begin
            order = 2'b00;
            light(red, amber, green, 0, order);
          end
        //任务的调用，交通灯控制时序
        always
            begin
              order = 2'b01;
              light(red, amber, green, RED_TICS, order);        //调用开灯任务，开红灯
              order = 2'b10;
              ilght(red, amber, green, GREEN_TICS, order);      //调用开灯任务，开绿灯
              order = 2'b11;
              light(red, amber, green, AMBER_TICS, order);      //调用开灯任务，开黄灯
            end
    endmodule
```

例中，程序中带有自己的时钟产生器，产生一个周期为 200、占空比为 1∶1 的时钟信号。模块的输出 red、amber 和 green 分别控制着红灯、黄灯和绿灯的开关。定义的任务 light 指定了交通灯开通某个颜色的时间，而输入变量 order 的值决定了所开通交通灯的颜色：若 order 的取值为 2'b00，则不开任何颜色的灯；如果 order 的取值为 2'b01，则开红灯；如果 order 的取值为 2'b10，则开绿灯；如果 order 的取值为 2'b11，则开黄灯。在程序中，repeat 循环语句用来控制亮灯的时间，输入变量 tic_time 的取值用来指定不同颜色的交通灯开通的时间，不同颜色的灯对应的 tic_time 值不一样。

5.5.2　函数

Verilog HDL 使用关键字 function 和 endfunction 对函数进行声明。对于一个子程序来说，如果下面的所有条件全部成立，则可以使用函数来完成：

(1) 子程序内不含有延时、时序或者事件控制结构。

(2) 子程序只有一个返回值。

(3) 至少有一个输入变量。

(4) 没有输出或者双向变量。

(5) 不含有非阻塞赋值语句。

Verilog HDL 的函数类似于其它编程语言中的函数，其目的都是通过函数运算返回一个值。与任务一样，函数也包括函数的定义与函数的调用。

1. 函数的定义

函数定义的语法格式如下：

```
function <返回值类型或位宽> <函数名>;
<输入参量与类型声明>
<局部变量声明>
    begin
            语句 1;
            语句 2;
            …
            语句 n;
     end
endfunction
```

说明：

(1) 函数定义是嵌入在关键字 function 和 endfunction 之间的，其中关键词 function 标志着一个函数定义结构的开端，endfunction 标志着一个函数定义结构的结束。"函数名"是给被定义函数取的名称，这个函数名在函数定义结构内部还代表着一个内部变量，函数调用后的返回值是通过这个函数名变量传递给调用语句的。

(2) "返回值类型或位宽"是一个可选项，用来对函数调用返回数据的类型或宽度进行说明，它可以有如下三种形式：

① [msb:lsb]：这种形式说明函数名所代表的返回数据变量是一个多位的寄存器变量，它的位宽由[msb:lsb]指定。比如函数定义语句

```
function [7:0] adder;
```

就定义了一个函数 adder，其函数名 adder 还代表着一个 8 位宽的寄存器变量，其中最高位为第 7 位，最低位为第 0 位。

② integer：这种形式说明函数名所代表的返回变量是一个整型变量。

③ real：这种形式说明函数名所代表的返回变量是一个实型变量。

如果在函数声明语句中缺省"返回值类型或位宽"这一项，则认为函数名所代表的返回值是一位寄存器类型的数据。

(3) "输入参量与类型声明"是对函数各个输入端口的宽度和类型进行声明。在函数定义中，至少必须有一个输入端口(input)的声明，不能有输出端口(output)的声明。数据类型声明语句用来对函数内用到的局部变量进行宽度和类型声明，该声明语句的语法与进行模块定义时的相应声明语句语法一致。

(4) "局部变量声明"是对函数内部局部变量进行宽度和类型的声明。

(5) 由 begin 与 end 关键词界定的一系列语句同任务中的一样，用来指明函数被调用时要执行的操作。在函数被调用时，这些语句将以串行方式得到执行。

例 5.5-4　统计输入数据中“0”的个数。

```
function[3:0] out0;
    input[7:0] x;
    reg[3:0] count;
    integer i;
        begin
          count = 0;
          for(i = 0; i <= 7; i = i+1)
                if(x[i] == 1'b0)        count = count+1;
          out0 = count;
        end
endfunction
```

例中定义了一个名为 out0 的函数，该函数有一个 8 位的输入端口 x，同时还定义了两个局部变量 i(整型变量)和 count(4 位寄存器变量)。

例中所定义的函数被调用时，begin 和 end 之间的行为语句得到执行，其中 for 循环语句实现了每检测到输入数据中某一位含 0，则将由 count 来计数。最后将得到的 count 值赋给函数名变量 out0。这样就可以通过这个函数名变量将结果值返回给调用语句。

函数定义时的注意事项：

(1) 与任务一样，函数定义结构只能出现在模块中，而不能出现在过程块内。

(2) 函数至少必须有一个输入端口。

(3) 函数不能有任何类型的输出端口(output 端口)和双向端口(inout 端口)。

(4) 在函数定义结构中的行为语句内不能出现任何类型的时间控制描述，也不允许使用 disable 终止语句。

(5) 与任务定义一样，函数定义结构内部不能出现过程块。

(6) 在一个函数内可以对其它函数进行调用，但是函数不能调用其它任务。

(7) 在第一行 function 语句中不能出现端口名列表。

(8) 函数声明的时候，在 Verilog HDL 的内部隐含地声明了一个名为 function_identifier(函数标识符)的寄存器类型变量，函数的输出结果将通过这个寄存器类型变量被传递回来。

2. 函数的调用

函数的调用是通过将函数作为表达式中的操作数来实现的。函数的调用格式如下：

<函数名> (<输入表达式 1>，<输入表达式 2>…<输入表达式 n>);

其中，“输入表达式”应与函数定义结构中说明的输入端口一一对应，它们代表着各个输入端口的输入数据。

函数调用时的注意事项：

(1) 函数的调用不能单独作为一条语句出现，它只能作为一个操作数出现在调用语句内。

(2) 函数的调用既能出现在过程块中，也能出现在 assign 连续赋值语句中。

(3) 函数定义中声明的所有局部寄存器都是静态的，即函数中的局部寄存器在函数的多个调用之间保持它们的值。

下面以阶乘运算为例来说明函数的调用方式。

例 5.5-5 阶乘函数示例。

```
module  tryfact_tb;
        function[31:0]factorial;                //函数的定义部分
                input[3:0]operand;
                reg[3:0]index;
                begin
                        factorial = 1;
                        for(index = 1; index <= operand; index = index+1)
                        factorial = index * factorial;
              end
              endfunction
              reg[31:0]result;
              reg[3:0]n;
              initial
                     begin
                        result = 1;
                        for(n = 1; n <= 9; n = n+1)
                        begin
                              result = factorial(n);    //函数的调用部分
                              $display("n = %d result = %d", n, result);
                        end
                     end
     endmodule
```

该例由函数定义和 initial 过程块构成，其中定义了一个名为 factorial 的函数，该函数是一个进行阶乘运算的函数，具有一个 4 位的输入端口，同时返回一个 32 位的寄存器类型的值；在 initial 块中定义了两个寄存器变量，分别为 32 位的 result 和 4 位的 n，initial 块对 1 至 9 进行阶乘运算，并打印出结果值。

```
n = 1    result =        1
n = 2    result =        2
n = 3    result =        6
n = 4    result =       24
n = 5    resul t=      120
n = 6    result =      720
n = 7    result =     5040
n = 8    result =    40320
n = 9    result =   362880
```

5.5.3 任务与函数的区别

在 Verilog HDL 中，任务和函数用于不同的目的，表 5.5-1 列出了任务和函数的区别。

任务和函数都必须在模块内进行定义，其作用范围仅局限于定义它们的模块。任务用于代替普通的 Verilog HDL 代码，其中可以包括延时、时序、事件等语法结构，并且可以具有多个输出变量。函数用于代替表示纯逻辑的 Verilog HDL 代码，在仿真时刻 0 就开始执行，只能有一个输出。因此函数一般用于完成各类转换和常用的计算。

任务可以有输入、输出和输入/输出(双向)变量，而函数只有输入变量。另外，可以在任务和函数中声明局部变量，如寄存器、整数、实数和时间，但是不能声明线网类型的变量。在任务和函数中只能使用行为级语句，但是不能包含 always 和 initial 块。设计者可以在 always 块、initial 块以及其它的任务和函数中调用任务和函数。

表 5.5-1 任务和函数的区别

函 数	任 务
函数能调用另一个函数，但不能调用另一个任务	任务能调用另一个任务，也能调用另一个函数
函数总是在仿真时刻 0 就开始执行	任务可以在非 0 仿真时刻执行
函数一定不能包含任何延时、时序或者事件控制声明语句	任务可以包含延时、时序或者事件控制声明语句
函数至少有一个输入变量	任务可以没有或者有多个输入(input)、输出(output)和双向(inout)变量
函数只能返回一个值，函数不能有输出(output)或者双向(inout)变量	任务不返回任何值，任务可以通过输出(output)或者双向(inout)变量传递多个值
函数不能单独作为一条语句出现，它只能以语句的一部分的形式出现	任务的调用是通过一条单独的任务调用语句实现的
函数调用可以出现在过程块或连续赋值语句中	任务调用只能出现在过程块中
函数的执行不允许由 disable 语句进行中断	任务的执行可以由 disable 语句进行中断

在 Verilog HDL 语法中，任务和函数是从高级程序语言中继承过来的，其语法使用范围具有一定的局限性，对于初学者而言掌握起来较为困难。实际上，在对数字电路的设计中 Verilog HDL 更倾向于使用模块来解决代码的重复问题，这种表示方法和电路的实际组成相近，在描述方式上也更加直观。

5.6 典型测试向量的设计

5.6.1 变量初始化

在 Verilog HDL 中，有两种方法可以初始化变量：一种是利用初始化变量；另一种是在定义变量时直接赋值初始化。这两种初始化任务是不可综合的，主要用于仿真过程。

1. initial 初始化方式

在大多数情况下，Testbench 中变量初始化的工作通过 initial 过程块来完成，可以产生

丰富的仿真激励。

initial 语句只执行一次，即在设计被开始模拟执行时开始(0 时刻)直到过程结束，专门用于对输入信号进行初始化和产生特定的信号波形。一个 Testbench 可以包含多个 initial 过程语句块，所有的 initial 过程都同时执行。

需要注意的是，initial 语句中的变量必须为 reg 类型。

例 5.6-1 利用 initial 初始化方式测试向量的产生。

```
module counter_demo2(cnt);
        output      [3:0]   cnt;
        reg         clk;
        reg          [3:0]   temp;
        initial      temp = 0;
        initial      clk = 0;
endmodule
```

2. 定义变量时初始化

在定义变量时初始化的语法非常简单，直接用“=”在变量右端赋值即可，例如：

```
reg [7:0] cnt = 8'b00000000;
```

就将 8 比特的寄存器变量 cnt 初始化为 0。

5.6.2 数据信号测试向量的产生

数据信号测试向量既可以通过 Verilog HDL 的时间控制功能(#、initial、always 语句)来产生各类验证数据，也可以通过系统任务来读取计算机上已存在的数据文件。

数据信号的产生有两种形式：其一是初始化和产生都在单个 initial 块中进行；其二是初始化在 initial 语句中完成，而产生在 always 语句块中完成。前者适合不规则数据序列，并且要求长度较短；后者适合具有一定规律的数据序列，长度不限。下面通过实例分别进行说明。

例 5.6-2 产生位宽为 4 的质数序列{1、2、3、5、7、11、13}，并且重复两次，其中样值间隔为 4 个仿真时间单位。

由于该序列无明显规律，因此利用 initial 语句最为合适，Verilog HDL 程序代码如下：

```
`timescale  1ns /1ps
module sequence_tb;
    reg         [3:0]       q_out;
    parameter   sample_period = 4;
    parameter   queue_num    = 2;
    initial begin
        q_out   = 0;
        repeat(queue_num) begin
          #   sample_period   q_out = 1;
          #   sample_period   q_out = 2;
```

```
            #   sample_period   q_out = 3;
            #   sample_period   q_out = 5;
            #   sample_period   q_out = 7;
            #   sample_period   q_out = 11;
            #   sample_period   q_out = 13;
        end
    end
endmodule
```

仿真结果示意图见图 5.6-1。

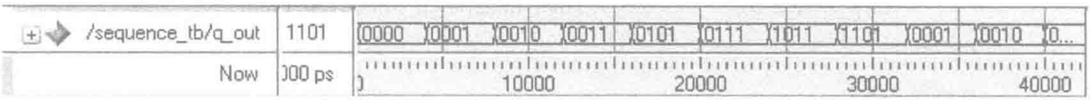


图 5.6-1　仿真结果示意图

5.6.3　时钟信号测试向量的产生

时序电路应用广泛，其中，时钟信号是时序电路设计最关键的参数之一，因此本节专门介绍如何产生仿真验证过程所需要的各类时钟信号。

例 5.6-3　产生占空比为 50%的时钟信号，其波形如图 5.6-2 所示。

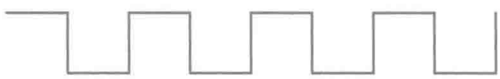


图 5.6-2　占空比为 50%的时钟信号

(1) 基于 initial 语句的方法。

```
module clk1(clk);
  output clk;
  parameter clk_period = 10;
  reg clk;
  initial begin
     clk = 0;
     forever    #(clk_period/2)   clk = ~clk;
  end
endmodule
```

(2) 基于 always 语句的方法。

```
module clk2(clk);
  output clk;
  parameter clk_period = 10;
  reg clk;
  initial     clk = 0;
  always # (clk_period/2)     clk = ~clk;
```

```
endmodule
```

initial 语句用于初始化 clk 信号，如果没有对 clk 信号的初始化赋值，会出现对未知信号取反的情况，从而造成 clk 信号在整个仿真阶段都为未知状态。

例 5.6-4　产生占空比可设置的时钟信号。

自定义占空比信号可以通过 always 模块快速实现。下面是占空比为 20%的时钟信号代码。

```
module Duty_Cycle(clk);
  output clk;
  parameter High_time = 5,
            Low_time = 20;    //占空比为 High_time/(High_time+Low_time)
  reg clk;
  always begin
        clk = 1;
        #High_time;
        clk = 0;
        #Low_time;
  end
endmodule
```

这里因为是直接对 clk 信号赋值，所以不需要 initial 语句初始化 clk 信号。

例 5.6-5　产生具有相位偏移的时钟信号。

相位偏移是两个时钟信号之间的相对概念，如图 5.6-3 所示。其中 clk_a 为参考信号，clk_b 为具有相位偏移的信号。

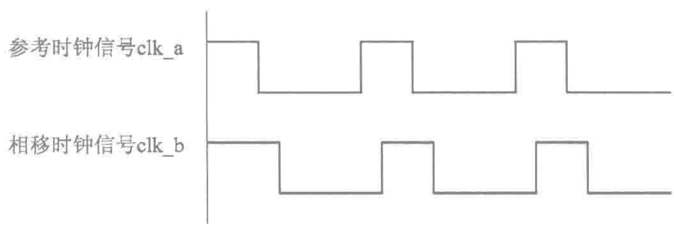


图 5.6-3　相位偏移时钟信号示意图

产生相移时钟信号的 Verilog HDL 程序代码如下：

```
module shift_Duty_Cycle(clk_a, clk_b);
  output clk_a, clk_b;
  parameter High_time = 5, Low_time = 5, pshift_time = 2;
  reg clk_a;
  wire clk_b;
  always begin
        clk_a = 1;
        # High_time;
        clk_a = 0;
```

```
            # Low_time;
        end
      assign # pshift_time clk_b = clk_a;
    endmodule
```

首先通过一个 always 模块产生参考时钟信号 clk_a，然后通过延时赋值得到 clk_b 信号，其偏移的相位可通过 360*pshift_time%(High_time+Low_time)来计算，其中%为取模运算。上述代码的相位偏移为 72°。

例 5.6-6 产生固定数目的时钟信号。

```
module fix_num_clk(clk);
  output clk;
  parameter clk_cnt = 5, clk_period = 2;
  reg clk;
  initial begin
    clk = 0;
    repeat (clk_cnt)
      # clk_period/2 clk = ~clk;
  end
endmoudle
```

上述代码产生了 5 个周期的时钟信号。

5.6.4 总线信号测试向量的产生

总线是运算部件之间数据流通的公共通道。在 RTL 级描述中，总线指的是由逻辑单元、寄存器、存储器、电路输入或其它总线驱动的一个共享向量。而总线功能模型则是一种将物理的接口时序操作转化成更高抽象层次接口的总线模型，如图 5.6-4 所示。

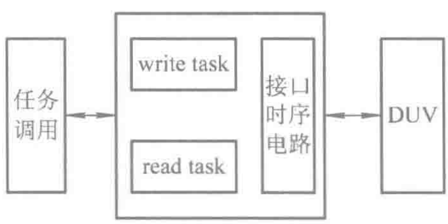


图 5.6-4 总线功能模型

在总线中，对于每个请求端，有一个输入来选择驱动该总线所对应的请求端。选择多个请求端时会产生总线冲突，根据不同的总线类型，会产生不同的冲突结果。当有多个请求端发出请求时，相应的操作由总线的类型决定。在 Verilog HDL 测试中，总线测试信号通常是通过将片选信号、读(或者写)使能信号、地址信号以及数据信号以 task 任务的形式来描述，并通过调用 task 形式的总线信号测试向量来完成相应的总线功能。

下面以工作频率为 100 MHz 的 AHB 总线写操作为例，说明以 task 形式产生总线信号测试向量的方法。图 5.6-5 是 AHB 总线写操作的时序图。其中，在完成数据的写操作后将

片选和写使能信号置为无效(低电平有效)。

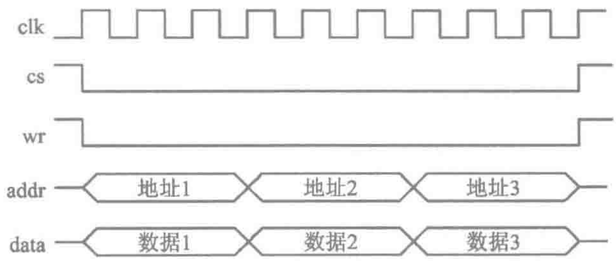


图 5.6-5　AHB 总线写操作时序图

例 5.6-7　产生一组具有写操作的 AHB 总线功能模型。

```
module bus_wr_tb;
  reg clk;
  reg cs;
  reg wr;
  reg [31:0] addr;
  reg [31:0] data;
  initial
      begin
         cs = 1'b1; wr = 1'b1;
         #30;
         bus_wr(32'h1100008a, 32'h11113000);
         bus_wr(32'h1100009a, 32'h11113001);
         bus_wr(32'h110000aa, 32'h11113002);
         bus_wr(32'h110000ba, 32'h11113003);
         bus_wr(32'h110000ca, 32'h11113004);
         addr = 32'bx; data = 32'bx;
      end
  initial clk = 1;
  always #5 clk = ~clk;

  task bus_wr;
  input [31:0] ADDR;
  input [31:0] DATA;
        begin
          cs = 1'b0; wr = 1'b0;
          addr = ADDR;
          data = DATA;
          #30 cs = 1'b1; wr = 1'b1;
        end
```

```
    endtask
endmodule
```

输出波形如图 5.6-6 所示。可以看出，在片选信号和写使能信号均有效时，每三个时钟周期输出一组地址和数据，当完成地址和数据的输出后，则将片选信号和写使能信号置为无效。

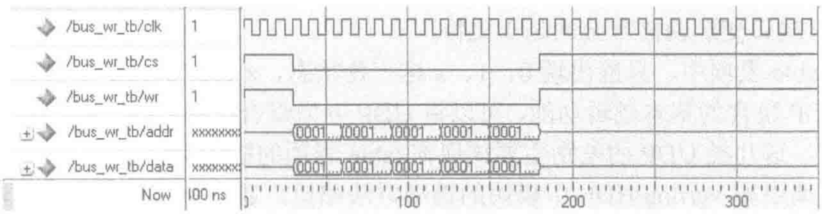


图 5.6-6　总线信号功能测试波形

5.7　用户自定义元件模型

Verilog HDL 中提供了一种扩展基元的方法，允许用户自己定义元件(User Defined Primitives，UDP)。通过 UDP，可以把一块组合逻辑电路或时序逻辑电路封装在一个 UDP 内，并把这个 UDP 作为一个基本门元件来使用。需要注意的是，UDP 不能综合，只能用于仿真。

5.7.1　UDP 的定义与调用

UDP 定义的语法和模块定义的类似，但由于 UDP 和模块属于同级设计，因此 UDP 定义不能出现在模块之内。模块定义使用由关键词 primitive 和 endprimitive 封装起来的一段代码，这段代码定义了该 UDP 的功能。这种功能的定义是通过表来实现的，即在这段代码中有一段处于关键词 table 和 endtable 之间的表，用户可以通过设置该表来规定 UDP 的功能。

UDP 的定义格式如下：

```
primitive <元件名称>(<输出端口名>，<输入端口名 1>，
                    <输入端口名 2>，…，<输入端口名 n>)；
   输出端口类型声明(output)；
   输入端口类型声明(input)；
   输出端口寄存器变量说明(reg)；
   元件初始状态说明(initial)；
    table
         <table 表项 1>；
         <table 表项 2>；
          …
         <table 表项 n>；
    endtable
endprimitive
```

与 Verilog HDL 中的模块相比，UDP 具有以下特点：

(1) UDP 的输出端口只能有一个，且必须位于端口列表的第一项。只有输出端口能定义为 reg 类型。

(2) UDP 的输入端口可有多个，一般时序电路 UDP 的输入端口最多 9 个，组合电路 UDP 的输入端口可多至 10 个。

(3) 所有端口变量的位宽必须是 1 比特。

(4) 在 table 表项中，只能出现 0、1、x 这三种状态，z 将被认为是 x 状态。

根据 UDP 包含的基本逻辑功能，可以将 UDP 分为组合电路 UDP、时序电路 UDP 及混合电路 UDP，这几类 UDP 的差别主要体现在 table 表项的描述上。

UDP 的调用和 Verilog HDL 中模块的调用方法相似，通过位置映射。其语法格式如下：

UDP 名 例化名(连接端口 1 信号名，连接端口 2 信号名，连接端口 3 信号名，…);

例 5.7-1 用 UDP 方式定义一个全加器仿真模型。

```
primitive summ(sum, cin, a, b);   //本位和 UDP
  output sum;
  input a, b, cin;
    table
    //cin  a  b  :  sum
        0  0  0  :  0;
        0  0  1  :  1;
        0  1  0  :  1;
        0  1  1  :  0;
        1  0  0  :  1;
        1  0  1  :  0;
        1  1  0  :  0;
        1  1  1  :  1;
    endtable
endprimitive
primitive carry(cout, cin, a, b);   //进位 UDP
  output cout;
  input a, b, cin;
    table
    //cin  a  b  :  cout
        0  0  0  :  0;
        0  0  1  :  0;
        0  1  0  :  0;
        0  1  1  :  1;
        1  0  0  :  0;
        1  0  1  :  1;
        1  1  0  :  1;
```

```
        1  1  1  :  1;
    endtable
endprimitive
```

例中定义了全加器本位和 UDP 与进位 UDP 的模型。其中，在进位 UDP 模型中，只要三个输入项(a、b 和 cin)中有两个值为 1，则 cout 的值一定为 1；如果输入中有两个值为 0，则 cout 的值也一定为 0。

5.7.2　UDP 应用实例

1. 组合电路 UDP 元件

组合逻辑电路的功能列表类似真值表，规定了不同的输入值和对应的输出值，表中每一行的形式是“output，input1，input2，…”，排列顺序与端口列表中的顺序相同。组合电路 UDP 的输入端口可多至 10 个。如果某个输入、输出组合没有定义输出，那么就把这种情况的输出置为 x。下面是一个单比特乘法器的 UDP 开发示例。

例 5.7-2　3 选 1 多路选择器的 UDP 开发示例。

```
primitive mux3_1(Y, in0, in1, in2, s2, s1);
  input in0, in1, in2, s2, s1;
  output Y;
  table
    //in0 in1 in2  s2  s1  :  Y
     0   ?   ?   0   0  :  0;          //当 s2s1 = 00 时，Y = in0
     1   ?   ?   0   0  :  1;
     ?   0   ?   0   1  :  0;          //当 s2s1 = 01 时，Y = in1
     ?   1   ?   0   1  :  1;
     ?   ?   0   1   ?  :  0;          //当 s2s1 = 1？ 时，Y = in2
     ?   ?   1   1   ?  :  1;
  endtable
endprimitive
```

其中，字符?代表不必关心相应变量的具体值，即它可以是 0、1 或 x。此外，Verilog HDL 标准规定，如果 UDP 输入端口中出现 z，则按照 x 处理。表 5.7-1 列出了 UDP 原语中可使用的值。

表 5.7-1　UDP 原语中可使用的值

符　号	意　义	符　号	意　义
0	逻辑 0	(AB)	由 A 变到 B
1	逻辑 1	*	与(??)相同
x	未知的值	r	上升沿，与(01)相同
?	0、1 或 x 中的任一个	f	下降沿，与(10)相同
b	0 或 1 中任选一个	p	(01)、(0x)和(x1)中的任一个
-	输出保持	n	(10)、(1x)和(x0)中的任一个

2. 时序电路 UDP 元件

UDP 还可以描述具有电平触发和边沿触发特性的时序电路。时序电路拥有内部状态序列，其内部状态必须用寄存器变量进行建模，该寄存器的值就是时序电路的当前状态，它的下一个状态是由放在基元功能列表中的状态转换表决定的，而且寄存器的下一个状态就是这个时序电路 UDP 的输出值。所以，时序电路 UDP 由两部分组成——状态寄存器和状态列表，定义时序电路 UDP 的工作也分为两部分——初始化状态寄存器和描述状态列表。

在时序电路的 UDP 描述中，01、0x、x1 代表着信号的上升沿。下面是一个上升沿 D 触发器的 UDP 开发示例。

例 5.7-3　通过 Verilog HDL 给出 D 触发器的 UDP 描述，并在模块中调用 UDP 组件。

```
primitive D_Edge_FF(Q, Clk, Data);
    output Q;
    reg   Q;
    input Data, Clk;
    initial Q = 0;
    table
      //Clk   Data : Q(Stata):   Q(next)
      (01)  0     :      ? :    0:
      (01)  1     :      ? :    1;
      (0x)  1     :      1 :    1;
      (0x)  0     :      0 :    0;
      (?0)  ?     :      ? :    -;                    //忽略时钟负边沿
      ?           (??)  :    ? :    -;               //忽略在稳定时钟上的数据变化
    endtable
endprimitive
```

表项(01)表示从 0 转换到 1，表项(0x)表示从 0 转换到 x，表项(?0)表示从任意值(0、1 或 x)转换到 0，表项(??)表示任意转换。对于任意未定义的转换，输出默认为 x。假定 D_Edge_FF 为 UDP 定义，则它现在就能够像基本门一样在模块中使用。

3. 混合电路 UDP 元件

在同一个表中能够混合电平触发项和边沿触发项。在这种情况下，边沿变化在电平触发之前处理，即电平触发项覆盖边沿触发项。下面是一段带异步清 0 D 触发器的 UDP 描述。

例 5.7-4　利用 Verilog HDL 完成异步清零 D 触发器的 UDP 描述。

```
primitive D_Async_FF(Q, Clk, Clr, Data);
    output Q;
    reg   Q;
    input Clr, Data, Clr;
     table                    //定义混合 UDP 元件
```

```
    //Clk Clr Data :(SQtate) :Q(next)
    (01) 0 0 : ? : 0 ;
    (01) 0 1 : ? : 1 ;
    (0x) 0 1 : 1 : 1 ;
    (0x) 0 0 : 0 : 0 ;
    (?0) 0 ? : ? : - ;              //忽略时钟负边沿
    (??) 0 ? : ? : - ;
    ? 1 ? : ? : 0 ;
    endtable
endprimitive
```

5.8　基本门级元件和模块的延时建模

为了有效验证数字集成电路中的功能，Verilog HDL 规定了模块和基本元件的延时模型。这样，设计者可以对所调用的门级或所设计的模块进行时序设计，完成对目标电路功能的验证。

5.8.1　门级延时建模

在实际的电路中，任何一个逻辑门都具有延时，信号从逻辑门的输入到输出的传输延时可以通过门延时来定义。用户可以通过门延时来说明逻辑电路中的延时，同时也可以指定输入端到输出端的延时。

门级延时可以分为如下四类：

(1) 上升延时：表示信号由 0、x 或 z 状态变化到 1 状态时受到的门传输延时。

(2) 下降延时：表示信号由 1、x 或 z 状态变化到 0 状态时受到的门传输延时。

(3) 到不定态的延时：表示信号由 0、1 或 z 状态变化到 x 状态时受到的门传输延时。

(4) 截止延时：表示信号由 0、1 或 x 状态变化到 z 状态时受到的门传输延时。

由于多输入门(and、nand、or、nor、xor 和 xnor)及多输出门(buf 和 not)的输出不可能是高阻态 z，所以这两类元件没有“到不定态的延时”这类门级延时，只能有“上升延时”、“下降延时”和“截止延时”这三种。对于三态门(bufif1、bufif0、notif1 和 notif0)，由于其输出可以取四种可能的逻辑状态(0、1、x 和 z)，所以具有全部门级延时种类。而对于上拉电阻(pullup)和下拉电阻(pulldown)，由于它们没有输入端口，所以不存在输入到输出的延时，因此就不存在任何形式的门级延时。

实例引用带延时参数门的语法格式如下：

```
gate_type #[delay] [instance_name] (terminal_list);
```

其中，delay 规定了门的延时，即从门的输入端到输出端的传输延时。若没有指定门延时值，则默认的延时值为 0。同时 delay 是由一个或多个延时值组成的，可以有基本延时表达和最小、典型、最大延迟表达两种形式。

1. 门级延时的基本延时表达形式

在门级延时的基本延时表达形式下，delay 内可以包含 0～3 个延时值，表 5.8-1 给出了指定的不同延时值个数时 delay 的 4 种表示形式。

表 5.8-1　延时参数形式

延时值	无延时	1 个延时值 (d)	2 个延时值 (d1, d2)	3 个延时值 (dA, dB, dC)
Rise	0	d	d1	dA
Fall	0	d	d2	dB
To_x	0	d	min(d1, d2)	min(dA, dB, dC)
Turn_off	0	d	min(d1, d2)	dC

说明：

(1) 当 delay 没有指定门延时值时，默认的延时值为 0。这意味着元件实例的“上升延时值”、“下降延时值”、“截止延时值”和“到不定态的延时值”均为 0。例如：

```
notif1 U0 (out, in, ctrl);
```

例中，门级延时值为 0，因为没有定义延时，所以元件实例 U0 的“上升延时值”、“下降延时值”、“截止延时值”和“到不定态的延时值”均为 0。

(2) 当 delay 内只包含 1 个延时值时，给定的延时值 d 将同时代表着元件实例的“上升延时值”、“下降延时值”、“截止延时值”和“到不定态的延时值”。例如：

```
notif1 #20 U1 (out, in, ctrl);
```

例中，门级延时值为 20，且只包含 1 个延时值，说明元件实例 U1 所有类型的门级延时都是 20 个单位时间。

(3) 当 delay 内包含了 2 个延时值时，元件实例的“上升延时值”由给定的 d1 指定，“下降延时值”由给定的 d2 指定，而“截止延时值”和“到不定态的延时值”将由 d1 和 d2 中的最小值指定。例如：

```
notif1 #(10, 20) U2 (out, in, ctrl);
```

例中，门级延时值为(10, 20)，包含了 2 个延时值 10 和 20，这表明元件实例 U2 将具有 10 个单位时间的“上升延时”和 20 个单位时间的“下降延时”，而它的“截止延时”和“到不定态的延时”将由 10 和 20 中的最小值指定，即 10 个单位时间。

(4) 当 delay 内包含了 3 个延时值时，元件实例的“上升延时值”由给定的 dA 指定，“下降延时值”由给定的 dB 指定，“截止延时值”由给定的 dC 指定，而它的“到不定态的延时值”将由 dA、dB 和 dC 中的最小值指定。例如：

```
notif1 #(10, 20, 30)U3 (out, in, ctrl);
```

例中，门级延时值为(10, 20, 30)，包含了 3 个延时值 10、20 和 30，这表明元件实例 U3 具有 10 个单位时间的“上升延时”、20 个单位时间的“下降延时”和 30 个单位时间的“截止延时”，而它的“到不定态的延时”将由 10、20 和 30 中的最小值指定，即 10 个单位时间。

2. 门级延时的“最小、典型、最大”延时表达形式

除了基本延时表达形式外，门级延时量还可以采用“最小、典型、最大”延时表达形式，采用这种表示方式时，门级延时量中的每一项将由“最小延时”、“典型延时”和“最大延时”三个值来表示，其语法格式如下：

```
#(d_min: d_typ: d_max)
```

采用“最小、典型、最大”延时表达形式时，delay 内可以包含 1～3 个延时值。例如：

```
#(dA_min: dA_typ: dA_max)
#(dA_min: dA_typ: dA_max, dB_min: dB_typ: dB_max)
#(dA_min: dA_typ: dA_max, dB_min: dB_typ: dB_max, dC_min: dC_typ: dC_max, )
```

例(1)：

```
and # (4: 5: 6) U1 (out, i1, i2);
```

其中，delay 中只包含 1 个延时值，其最小值为 4、典型值为 5、最大值为 6。元件实例 U1 的“上升延时值”、“下降延时值”、“截止延时值”和“到不定态的延时值”如表 5.8-2 所示。

表 5.8-2 1 个延时值结果

最小延时值	上升延时值 = 4	下降延时值 = 4	到不定态的延时值 = 4	截止延时值 = 4
典型延时值	上升延时值 = 5	下降延时值 = 5	到不定态的延时值 = 5	截止延时值 = 5
最大延时值	上升延时值 = 6	下降延时值 = 6	到不定态的延时值 = 6	截止延时值 = 6

例(2)：

```
and # (3: 4: 5, 5: 6: 7) U2 (out, i1, i2);
```

其中，delay 中包含了 2 个延时值，第一个延时值的最小值为 3、典型值为 4、最大值为 5，第二个延时值的最小值为 5、典型值为 6、最大值为 7。元件实例 U2 的“上升延时值”由第一个延时值指定，“下降延时值”由第二个延时值指定，“到不定态的延时值”和“截止延时值”均由两个延时值中的最小值指定。各值的取值情况如表 5.8-3 所示。

表 5.8-3 2 个延时值结果

最小延时值	上升延时值 = 3	下降延时值 = 5	到不定态的延时值= min(3, 5)	截止延时值= min(3, 5)
典型延时值	上升延时值 = 4	下降延时值 = 6	到不定态的延时值= min(4, 6)	截止延时值= min(4, 6)
最大延时值	上升延时值 = 5	下降延时值 = 7	到不定态的延时值= min(5, 7)	截止延时值= min(5, 7)

例(3)：

```
and # (2: 3: 4, 3: 4: 5, 4: 5: 6) U3 (out, i1, i2);
```

其中，delay 中包含了 3 个延时值，第一个延时值的最小值为 2、典型值为 3、最大值为 4，第二个延时值的最小值为 3、典型值为 4、最大值为 5，第三个延时值的最小值为 4、典型值为 5、最大值为 6。元件实例 U3 的“上升延时值”由第一个延时值指定，“下降延时值”由第二个延时值指定，“截止延时值”由第三个延时值指定，而它的“到不定态的延时值”由三个延时值中的最小值指定。各值的取值情况如表 5.8-4 所示。

表 5.8-4　3 个延时值结果

最小延时值	上升延时值 = 2	下降延时值 = 3	到不定态的延时值= min(2, 3, 4)	截止延时值 = 4
典型延时值	上升延时值 = 3	下降延时值 = 4	到不定态的延时值= min(3, 4, 5)	截止延时值 = 5
最大延时值	上升延时值 = 4	下降延时值 = 5	到不定态的延时值= min(4, 5, 6)	截止延时值 = 6

例 5.8-1　用 Verilog HDL 建立图 5.8-1 模块 D 的延时仿真模块。

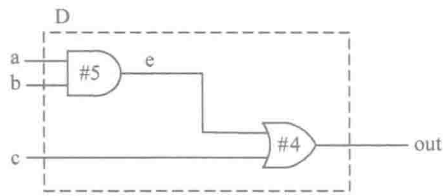


图 5.8-1　模块 D

图 5.8-1 逻辑电路的功能是 out = (a*b) + c。

其门级实现如图所示，其中包含了一个延迟时间为 5 个单位时间的与门和一个延迟时间为 4 个单位时间的或门。

带有延时的模块 D 的 Verilog HDL 程序代码如下：

```
module D(out, a, b, c);
  output out;
  input a, b, c;
  wire e;
  and #(5) a1(e, a, b);              //延迟 5 个单位时间
  or   #(4) o1(out, e, c);           //延迟 4 个单位时间
endmodule
```

带有延时的模块 D 的测试激励模块：

```
module D_tb;
  reg A, B, C;
  wire OUT;
  D d1 (.out(OUT), .a(A), .b(B), .c(C));

  initial
    begin
          A = 1'b0; B = 1'b0; C = 1'b0;
      #10 A = 1'b1; B = 1'b1; C = 1'b1;
      #10 A = 1'b1; B = 1'b0; C = 1'b0;
      #20 $finish;
    end
endmodule
```

其仿真波形如图 5.8-2 所示。

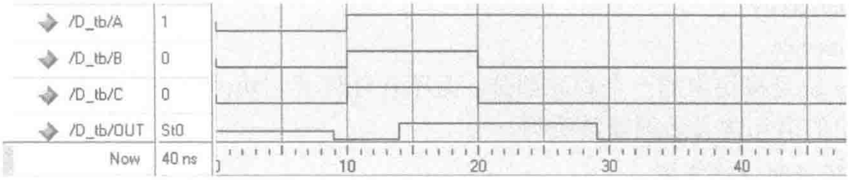


图 5.8-2　模块 D 的仿真波形

5.8.2　模块延时建模

对于由用户自己定义设计的模块，可以采用“加入门级延时说明”的办法，通过“延时说明块(Specify Block)”的结构来对模块的传输延时进行说明。注意，延时说明块既可以出现在行为描述模块内，也可以出现在结构描述模块内。

1. 延时说明块

在模块输入和输出引脚之间的延时称为模块路径延时。在 Verilog HDL 中，在关键字 specify 和 endspecify 之间给路径延时赋值，关键字之间的语句组成 specify 块(即指定块)。specify 与 endspecify 分别是延时说明块的起始标识符和终止标识符。

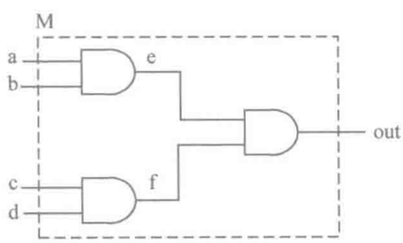
specify 块中包含下列操作语句：

(1) 定义穿过模块的所有路径延时；

(2) 在电路中设置时序检查；

(2) 定义 specparam 常量。

例 5.8-2　以图 5.8-3 为例，用 specify 块来描述图中 M 模块的路径延时。



路径	延时
a--e--ut	9
b--e--out	9
c--f--out	11
d--f--out	11

图 5.8-3　模块路径延时

Verilog HDL 程序代码如下：

```
module M (out, a, b, c, d);
  input a, b, c, d;
  output out;
  wire e, f;
  assign out = (a&b) | (c&d);         //逻辑功能
  specify                             //包含路径延时语句的 specify 块
    (a => out) = 9;
    (b => out) = 9;
    (c => out) = 11;
    (d => out) = 11;
```

```
    endspecify
endmodule
```

specify 块是模块中的一个独立部分，且不在任何其它块(如 initial 或 always)内出现。specify 块中的语句含义必须非常明确。

2. 路径延时描述方式

1) 并行连接

每条路径延时语句都有一个源域或一个目标域。在例 5.8-2 的路径延时语句中，a、b、c 和 d 在源域位置，而 out 是目标域。

在 specify 块中，用符号“=>”说明并行连接，其语法格式如下：

```
(<source_field> => <destination_field>) = <delay_value>;
```

其中，<delay_value>可以包含 1～3 个延时量，也可以采用“最小、典型、最大”延时表达形式。在延时量由多个值组成的情况下，应在延时量的外面加上一对括号。例如：

```
(a=>out) = (8:9:10);
```

表示的是输入 a 到输出 b 的最小、典型和最大延时分别是 8、9、10 个时间单位。

在并行连接中，源域中的每一位与目标域中相应的位连接。如果源域和目标域是向量，必须有相同的位数，否则会出现不匹配。因此，并行连接说明了源域的每一位到目标域的每一位之间的延时。

图 5.8-4 显示了源域和目标域之间的位是如何并行连接的。例 5.8-3 则给出了并行连接的 Verilog HDL 描述。

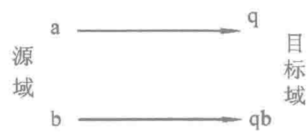


图 5.8-4　并行连接

例 5.8-3　并行连接定义延时示例。

```
module parallel_connected(out, a, b);
  input a, b;
  output out;
  wire out;
    assign out = a&b;   //逻辑功能
    specify
          (a => out) = 9;
          (b => out) = 11;
    endspecify
endmodule
```

例中描述了两条并行连接的路径延时，其中定义了端口 a 到端口 out 的路径延迟时间为 9 个时间单位，端口 b 到端口 out 的路径延迟时间为 11 个时间单位。

2) 全连接

在 specify 块中，用符号“*>”表示全连接，其语法格式如下：

```
(<source_field>*><destination_field>) = <delay_value>;
```

在全连接中，源域中的每一位与目标域中的每一位相连接。如果源域和目标域是向量，则它们的位数不必相同。全连接描述源域中的每一位和目标域中的每一位之间的延时，如图 5.8-5 所示。

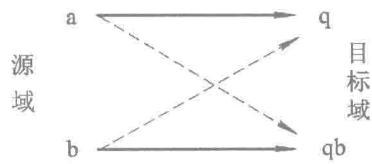


图 5.8-5　全连接

例 5.8-4　全连接定义延时示例。

```
module full_connected(out, a, b, c, d);
  input a, b, c, d;
  output out;
  wire out;
      assign out = (a&b) | (c&d);   //逻辑功能
      specify
        (a, b *> out) = 9;
        (c, d *> out) = 11;
      endspecify
endmodule
```

该例描述了四条全连接的路径延时，其中定义了端口 a 和端口 b 到端口 out 的路径延时为 9 个时间单位，端口 c 和端口 d 到端口 out 的路径延时为 11 个时间单位。

3. specparam 声明语句

specparam 用来定义 specify 块中的参数，例 5.8-5 是一个使用 specparam 语句的 specify 块示例。

例 5.8-5　specparam 声明语句示例。

```
module parallel_connected(out, a, b);
  input a, b;
  output out;
  wire out;
    assign out = a&b;   //逻辑功能
    specify
          //在指定块内部定义参数
          specparam a_to_out = 9;
          specparam b_to_out = 11;
          (a => out) = a_to_out;
          (b => out) = b_to_out;
    endspecify
endmodule
```

specparam 语句的格式和作用都类似于 parameter 参数说明语句，但两者又有不同：

(1) specparam 语句只能在延时说明块(specify 块)中出现，而 parameter 语句则不能在延时说明块内出现。

(2) 由 specparam 语句进行定义的参数只能是延时参数，而由 parameter 语句定义的参

数可以是任何数据类型的常数参数。

(3) 由 specparam 语句定义的延时参数只能在延时说明块内使用，而 parameter 语句定义的参数则可以在模块内的任意位置使用。

在模块中提供 specify 参数是为了方便给延时赋值，因此建议用 specify 参数而不是数值来表示延时。这样，如果电路的时序说明变化了，则用户只需要改变 specify 的参数值，而不必逐个修改每条路径的延时值。

5.8.3　与时序检查相关的系统任务

Verilog HDL 提供了系统任务来进行时序检查，该类任务很多，表 5.8-5 列举了部分与时序检查相关的系统任务。

表 5.8-5　与时序检查相关的系统任务

名　称	语 法 格 式	含　义
$setup	$setup(data_event, reference_event, limit);	如　果 (time_of_reference_event-time_of_data_event)< limit，则报告时序冲突(timing violation)
$hold	$hold(reference_event, data_event, limit);	如果(time_of_data_event-time_of_reference_event) <limit，则报告数据保持时间时序冲突
$setuphold	$setuphold(reference_event, data_event, setup_limit, hold_limit);	如果(time_of_data_event-time_of_reference_event) < setup_limit(或 hold_limit)，则报告数据保持时间时序冲突
$width	$width(reference_event, limit, threshold);	如果 threshold <(time_of_data_event-time_of_reference _event)<limit，则报告信号上出现脉冲宽度不够宽的时序错误
$period	$period(reference_event, limit);	检查信号的周期，若(time_of_data_event-time_of_reference_event) < limit，则报告时序错误
$skew	$skew(reference_event, data_event, limit);	检查信号之间(尤其是成组的时钟控制信号之间)的偏斜(skew)是否满足要求，若 time_of_data_event-time_of_reference_event > limit，则报告信号之间出现时序偏斜太大的错误
$recovery	$recovery(reference_event, data_event, limit);	若 (time_of_data_event - time_of_reference_event) < limit，则报告时序冲突
$nochange	$nochange(reference_event, data_event, start_edge_offset, end_edge_offset);	如果在指定的边沿触发基准事件区间发生数据变化，则报告时序冲突错误

下面介绍两种最常用的时序检查系统任务 $setup 和 $hold。所有的时序检查只能用在 specify 块里。为了简化讨论，此处省略了这些时序检查系统任务的一些可选参数。

1) $setup

建立时间检查可以使用系统任务 $setup。其语法格式如下：

```
$setup(data_event, reference_event, limit);
```

其中，data_event 是被检查的信号，要检查它是否违反约束；reference_event 用于检查

data_event 信号的参考信号；limit 是 data_event 需要的最小建立时间。如果(time_of_reference_event - time_of_data_event) < limit，则报告时序冲突。例如：

```
specify
    $setup (data, posedge clock, 3);
endspecify
```

该程序的功能是建立时间检查，clock 为参考信号，data 是被检查的信号，如果 time_posedge_clock - time_data < 3，则报告违反约束。

2) $hold

保持时间检查可以使用系统任务$hold。其语法格式如下：

```
$hold (reference_event, data_event, limit);
```

其中，reference_event 用于检查 data_event 信号的参考信号；data_event 是被检查的信号，要检查它是否违反约束；limit 是 data_event 需要的最小保持时间。如果(time_of_data_event - time_of_reference_event) < limit，则报告数据保持时间时序冲突。例如：

```
specify
    $hold (posedge clock, data, 5);
endspecify
```

该程序的功能是保持时间检查，clock 为参考信号，data 是被检查的信号，如果 time_data - time_posedge_clk < 5，则报告违反约束。

5.9 编译预处理语句

编译预处理是 Verilog HDL 编译系统的一个组成部分，指编译系统会对一些特殊命令进行预处理，然后将预处理结果和源程序一起再进行通常的编译处理。以“`”(反引号)开始的某些标识符是编译预处理语句。在 Verilog HDL 编译时，特定的编译指令在整个编译过程中有效(编译过程可跨越多个文件)，直到遇到其它不同的编译程序指令。常用的编译预处理语句如下：

(1) `define, `undef;

(2) `include;

(3) `timescale;

(4) `ifdef, `else, `endif;

(5) `default_nettype;

(6) `resetall;

(7) `unconnect_drive, `nounconnected_drive;

(8) `celldefine, `endcelldefine。

5.9.1 宏定义

`define 指令是一个宏定义命令，通过一个指定的标识符来代表一个字符串，可以增加 Verilog HDL 代码的可读性和可维护性，找出参数或函数不正确或不允许的地方。

`define 指令类似 C 语言中的#define 指令，可以在模块的内部或外部定义，编译器在编译过程中遇到该语句将把宏文本替换为宏的名字。`define 的声明语法格式如下：

```
`define <macro_name><Text>
```

对于已声明的语句，在代码中的应用格式如下(不要漏掉宏名称前的“`”)：

```
`macro_name
```

例如：

```
`define MAX_BUS_SIZE 32
…
reg [`MAX_BUS_SIZE – 1:0] AddReg;
```

一旦`define 指令被编译，其在整个编译过程中都有效。例如，通过另一个文件中的`define 指令，MAX_BUS_SIZE 能被多个文件使用。

`undef 指令用于取消前面定义的宏。例如：

```
`define   WORD 16          //建立一个文本宏替代
…
wire [`WORD : 1]Bus;
…
`undef   WORD              //在`undef 编译指令后，WORD 的宏定义不再有效
```

宏定义指令的注意事项：

(1) 宏定义的名称可以是大写，也可以是小写，但要注意不要和变量名重复。

(2) 和所有编译器伪指令一样，宏定义在超过单个文件边界时仍有效(对工程中的其它源文件)，除非被后面的`define、`undef 或`resetall 伪指令覆盖，否则`define 不受范围限制。

(3) 当用变量定义宏时，变量可以在宏正文中使用，并且在使用宏的时候可以用实际的变量表达式代替。

(4) 通过用反斜杠“\”转义中间换行符，宏定义可以跨越几行，新的行是宏正文的一部分。

(5) 宏定义行末不需要添加分号表示结束。

(6) 宏正文不能分离的语言记号包括注释、数字、字符串、保留的关键字、运算符。

(7) 编译器伪指令不允许作为宏的名字。

(8) 宏定义中的文本也可以是一个表达式，并不仅仅用于变量名称的替换。

`define 和 parameter 是有区别的。`define 和 parameter 都可以用于完成文本替换，但其存在本质上的不同，前者是编译之前就预处理，而后者是在正常编译过程中完成替换的。此外，`define 和 parameter 存在下列两点不同之处：

(1) 作用域不同。parameter 作用于声明的那个文件；`define 从编译器读到这条指令开始到编译结束都有效，除非遇到 `undef 命令使之失效，可以应用于整个工程。如果要让 parameter 作用于整个项目，可以将声明语句写于单独文件中，并用`include 让每个文件都包含声明文件。

`define 可以写在代码的任何位置，而 parameter 则必须在应用之前定义。通常编译器都可以定义编译顺序，或者从最底层模块开始编译，因此写在最底层就可以了。

(2) 传递功能不同。parameter 可以用作模块例化时的参数传递，实现参数化调用；`define

语句则没有此作用。`define 语句可以定义表达式，而 parameter 只能定义变量。

例 5.9-1 `define 使用示例。

```
module define_demo(clk, a, b, c, d, q)
  `define bsize 9
  `define c a+b
  input   clk;
  input [`bsize:0] a, b, c, d;
  output [`bsize:0] q;
  reg [`bsize:0] q;
      always @(posedge clk)begin
            q <= `c+d;
      end
endmodule
```

5.9.2 文件包含处理

所谓文件包含处理，是一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。Verilog HDL 提供了`include 命令用来实现“文件包含”的操作。其一般形式如下：

```
`include "文件名"
```

图 5.9-1 表示“文件包含”。图(a)为文件 File1.v，它有一个“`include "File2.v"”命令，还有其它的内容(以 A 表示)。图(b)为文件 File2.v，文件的内容以 B 表示。在编译预处理时，要对`include 命令进行“文件包含”预处理：将 File2.v 的全部内容复制插入到“`include "File2.v"”命令出现的地方，即 File2.v 被包含到 File1.v 中，得到图(c)所示的结果。在接着进行的编译中，将“包含”以后的 File1.v 作为一个源文件单位进行编译。

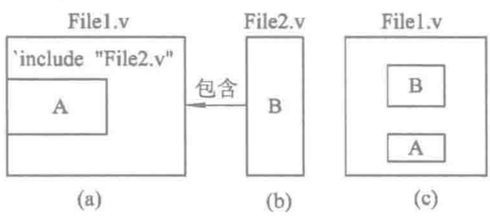


图 5.9-1 文件包含

“文件包含”命令可以节省程序设计人员的重复劳动，即可以将一些常用的宏定义命令或任务(task)组成一个文件，然后用`include 命令将这些宏定义包含到自己所写的源文件中，这相当于工业上对标准元件的使用。另外，在编写 Verilog HDL 源文件时，一个源文件可能经常要用到另外几个源文件中的模块，这种情况下即可用`include 命令将所需模块的源文件包含进来。

例 5.9-2 “文件包含”使用示例。

文件 a1.v：

```
module aaa(a, b, out);
```

```
        input a, b;
        output out;
        wire out;
            assign    out = a^b;
    endmodule
```

文件 b1.v：

```
    `include    "a1.v"
    module    b1(c, d, e, out);
        input    c, d, e;
        output    out;
        wire    out_a;
        wire    out;
            a1    U1(.a(c), .b(d), .out(out_a));
            assign    out = e&out_a;
    endmodule
```

例中，文件 b1.v 用到了文件 a1.v 中的模块 a1 的实例器件，通过“文件包含”处理来调用。模块 a1 实际上是作为模块 b1 的子模块被调用的。

5.9.3 仿真时间标度

`timescale 命令用来说明跟在该命令后的模块的时间单位和时间精度。其语法格式如下：

```
`timescale<时间单位>/<时间精度>
```

其中，“时间单位”用来定义模块中仿真时间和延迟时间的基准单位。“时间精度”用来声明该模块的仿真时间的精确程度，该参量被用来对延迟时间值进行取整操作(仿真前)，因此该参量又被称为取整精度。如果在同一个程序设计中存在多个`timescale 命令，则用最小的时间精度值来决定仿真的时间单位。另外，时间精度至少要和时间单位一样精确，时间精度值不能大于时间单位值。

在`timescale 命令中，用于说明时间单位和时间精度参量值的数字必须是整数，其有效数字为 1、10、100，单位为秒(s)、毫秒(ms)、微秒(μs)、纳秒(ns)、皮秒(ps)、毫皮秒(fs)。下面举例说明`timescale 命令的用法。

例 5.9-3 仿真时间标度示例。

例(1)：

```
`timescale    1ns/1ps
```

表示模块中所有的时间值都是 1 ns 的整数倍。这是因为在`timescale 命令中，定义了时间单位是 1 ns。因为`timescale 命令定义的时间精度为 1 ps，所以模块中的延迟时间可表达为带三位小数的实型数。

例(2)：

```
`timescale 10us/100ns
```

该例中，模块中的时间值均为 10 μs 的整数倍。`timesacle 命令定义的时间单位是 10 μs，延迟时间的最小分辨度为十分之一微秒(100 ns)，即延迟时间可表达为带一位小数的实型数。

例(3)：

```
`timescale 10ns/1ns
module  delay_tb;
    reg  set;
    parameter  d = 1.55;
    initial
      begin
        #d set = 0;
        #d set = 1;
      end
endmodule
```

该例中，`timescale 命令定义了模块 delay_tb 的时间单位为 10 ns、时间精度为 1 ns。因此在此测试模块中，所有的时间值应为 10 ns 的整数倍，且以 1 ns 为时间精度。这样经过取整操作，存在参数 d 中的延迟时间实际是 16 ns(即 1.6×10 ns)，这意味着在仿真时间为 16 ns 时寄存器 set 被赋值 0，在仿真时间为 32 ns 时寄存器 set 被赋值 1。

5.9.4　条件编译

一般情况下，Verilog HDL 源程序中所有的行都参加编译。但有时希望其中的一部分内容只有在满足条件时才对其进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。也就是说，当满足条件时对一组语句进行编译，而当条件不满足时则编译另一部分。

条件编译命令有以下几种形式。

形式 1：

```
`ifdef 宏名(标识符)
程序段 1
`else
程序段 2
`endif
```

作用：若“宏名”已经被定义过(用`define 命令定义)，则对程序段 1 进行编译，程序段 2 将被忽略；否则编译程序段 2，程序段 1 被忽略。其中`else 部分可以没有，即成为形式 2。

形式 2：

```
`ifdef 宏名 (标识符)
程序段 1
`endif
```

这里，“宏名”是一个 Verilog HDL 的标识符，“程序段”可以是 Verilog HDL 语句组，也可以是命令行。这些命令可以出现在源程序的任何地方。

注意：被忽略掉不进行编译的程序段部分也要符合 Verilog HDL 程序的语法规则。

通常在 Verilog HDL 程序中用到`ifdef、`else、`endif 编译命令的情况有以下几种：

- 选择一个模块的不同代表部分。
- 选择不同的时序或结构信息。
- 对不同的 EDA 工具，选择不同的激励。

5.9.5 其它语句

除了上述常用的编译预处理语句外，Verilog HDL 还包括其它的预处理语句，但由于其应用范围并不广泛，因此这里只对其进行简单介绍。

1) `default_nettype 语句

`default_nettype 为隐式线网类型，也就是将那些没有被说明的连线定义为线网类型。例如：

```
`default_nettype wand
```

该例定义的默认的线网为线与类型。因此，如果在此指令后面的任何模块中含有未说明的连线，那么该线网被假定为线与类型。

2) `resetall 语句

`resetall 编译器指令用于将所有的编译指令重新设置为默认值。例如：

```
`resetall
```

该单独的指令使得默认连线类型为线网类型。

3) `unconnected_drive 语句

在模块实例化中，将出现在`unconnected_drive 和`nounconnected_drive 两个编译指令之间的任何未连接的输入端口设置为正偏电路状态或反偏电路状态。例如：

```
`unconnect_drive pull1
/*在这两个程序指令间的所有未连接的输入端口为正偏电路状态(连接到高电平)*/
`nounconnected_drive
`unconnected_drive pull0
/*在这两个程序指令间的所有未连接的输入端口为反偏电路状态(连接到低电平)*/
`nounconnected_drive
```

4) `celldefine 语句

`celldefine 和`endcelldefine 这两个程序指令用于将模块标记为单元模块，表示包含模块定义。例如：

```
`celldefine
module FD1S3AX(D, CK, Z);
...
endmodule
`endcelldefine
```

5.10 Verilog HDL 测试方法简介

在集成电路的设计过程中，在完成所规定的电路设计之后，为了验证其功能的正确性，还需要针对所设计的电路编写相应的测试程序。测试程序通过调用被测试电路，将产生的

输入激励信号加载到被测试电路中，经过电路的逻辑功能处理输出相应的仿真结果，设计者通过仿真结果对电路功能进行判断，确认电路功能的正确性。

在集成电路测试领域，常用的测试方法有完全测试法、随机测试法和自动测试法。

1. 完全测试法

对于复杂的设计来说，常常通过检查代码的覆盖率来检查验证工作是否完成。代码覆盖率可以指示 Verilog HDL 代码描述的功能有多少在仿真过程中被验证过。通常代码覆盖率包括以下内容：

- 语句覆盖率；
- 路径覆盖率；
- 状态机覆盖率；
- 触发覆盖率；
- 表达式覆盖率。

2. 随机测试法

Verilog HDL 提供了多个用于随机测试的系统命令，通常使用随机测试的系统函数来仿真真实应用的情况，如在通信领域常用的帧同步搜索电路需要从接收的数据流中检测发送端固定插入的某个特殊的码型，而数据本身也有可能包括该码型，在这种情况下进行随机化测试就更接近于真实应用的情况。其中最常用的系统命令是随机数产生系统任务 $random。

3. 自动测试法

在集成电路测试领域，全面准确地测试才能保证大规模集成电路的正常工作，而确保一个设计能够得到全面测试的唯一途径就是实现任务的自动化。通常通过创建一个检验表，使用相应个数的采样值来完成自动测试。当修改过源代码后，所有的测试程序都自动被再次执行。但需要注意的是，使用自动测试可能会存在截断误差。

本章小结

随着设计规模的不断扩大，验证任务在设计中所占的比例越来越大，已成为 Verilog HDL 设计流程中非常关键的一个环节。如何编写高效、简洁的测试代码来实现验证任务，实现对设计模块功能正确性的验证，正成为验证这一关键环节的重要手段。

Verilog HDL 仿真和验证是与电路设计同等重要的功能，具有独特的设计方法，本章从电路仿真模型、系统函数、电路延时模型设计、任务和函数、信号激励产生、编译指令等方面深入地介绍了其设计方法。

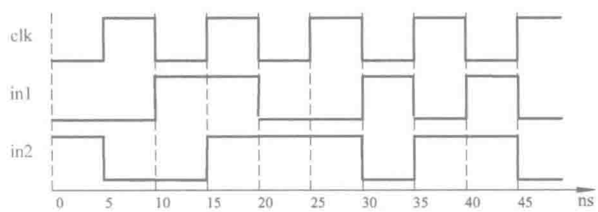
思考题和习题

1. 什么是验证？
2. 什么是仿真？

3. Testbench 是测试激励吗？测试程序的一般结构是什么？
4. 如何提高仿真效率？举例说明。
5. 下列程序的显示结果是什么？

```
module shifttest;
    reg[3:0] a;
    initial
       begin
         a = 4'b1001;
         $display(a>>1);
       end
endmodule
```

6. $finish(n)和$stop(n)中的参数 n 取不同的值时，分别表示什么含义？
7. 在 Verilog HDL 中，如何从文件中读入数据？
8. 试用 Verilog HDL 产生下图所示的测试信号。

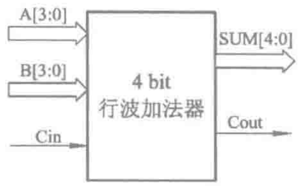


9. 根据下面的程序，画出产生的信号波形。

```
module para (a, b);
    output a, b;
    reg a, b;
    initial
       begin
         a = 1;
         b = 1;
         #100 a = 0;
         fork
           b = 0;
           #50 b = 1;
           #150 a = 1;
         join
         #100 b = 0;
         #50 a = 0;
         b = 1;
       end
endmodule
```

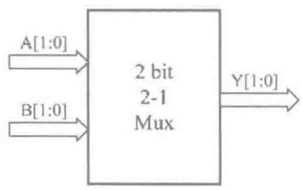
10. 什么是 Verilog HDL 中的任务(task)? 任务的定义方式和调用方式分别是什么?

11. 用任务方式编写下图所示的 4 bit 行波加法器，再调用此任务完成 8 bit 行波进位加法器。



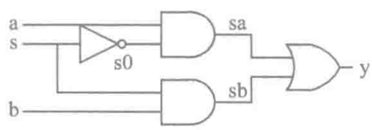
12. 什么是 Verilog HDL 中的函数(function)? 函数的定义方式和调用方式分别是什么?

13. 用函数编写下图所示的 2 bit 2-1 多路输入选择器，再调用此函数完成 2 bit 4-1 多路选择器。



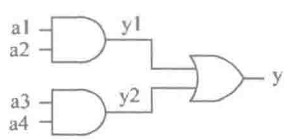
14. 如果需要设计并行激励，应使用什么语言元素?

15. 如下图所示电路，若其延迟时间设定如表中所示，试编写 Verilog HDL 程序设计该电路。



路　径	最小值(min)	典型值(type)	最大值(max)
a_sa_y	10	12	14
s_s0_sa_y	15	17	19
s_sb_y	11	13	15
b_sb_y	10	12	14

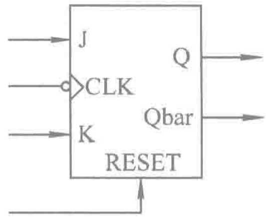
16. 用并行连接方式规划下图，再依表设计该电路。



输　入	输　出	上升延时	下降延时
a1	y	12	10
a2	y	12	10
a3	y	8	6
a4	y	8	6

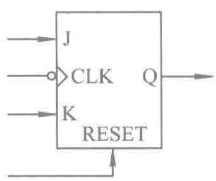
17. 用全连接方式重新设计习题 16。

18. 如下图所示，若其延迟时间如表中所列，用全连接方式设计该电路。



输入	输出	延迟时间
J	Q	8
J	Qbar	8
CLK	Q	10
CLK	Qbar	12
K	Q	8
K	Qbar	8
RESET	Q	4
RESET	Qbar	5

19. 如下图所示 J-K 触发器，其延迟时间如表中所示，试编写 Verilog HDL 程序设计该电路。



延迟时间	最小值(min)	典型值(type)	最大值(max)
上升延时 t_rise_LCK_Q	14	16	18
下降延时 t_fall_CLK_Q	17	19	21
关断延时 t_turnoff_CLK_Q	18	20	22

20. 简述`define 和 parameter 的区别。

21. 用 Verilog HDL 为例 4.3-8 的环形移位寄存器编写测试平台程序，并用系统任务 $display 把当前时间显示出来。

22. 用 Verilog HDL 为例 4.3-9 的序列信号发生器编写测试平台程序。

23. 用 Verilog HDL 为例 4.4-1 的顺序脉冲发生器编写测试平台程序。

24. 用 Verilog HDL 为例 4.4-2 的自动售报机编写测试平台程序。

第 6 章 Verilog HDL 高级程序设计举例

6.1 数字电路系统设计的层次化描述方法

集成电路设计中大量采用的是结构性的描述方法，归纳起来主要有两种：自下而上(Bottom-Up)的设计方法与自上而下(Top-Down)的设计方法。在实际运用中，可以根据实际设计的情况选择这两种方法相结合的设计方法，即混合设计方法。Top-Down 方法主要是从系统设计的角度进行，系统工程师往往会在项目的规划阶段将数字电路系统进行划分，明确主要单元模块的功能、时序和接口参数等系统方案，这种方法可以有效协调 ULSI 和 VLSI 芯片设计中众多开发人员的工作。Bottom-Up 方法主要是在实际系统执行过程中，底层设计人员提出底层功能模块的优化方案，系统设计人员分析这些优化对系统的影响，通过调整系统结构，提高芯片整体性能。

1. Bottom-Up 设计方法

Bottom-Up 设计方法是一种传统的设计方法，它要求电路设计者将系统进行模块划分，从底层模块设计开始，运用各底层模块搭建一个完整的系统。在这种设计方法中，首先根据系统设计的要求，定义并建立所需要的叶子模块，通过模块连接方式建成较大的模块，然后把这些比较大的模块组合成具有一定功能的模块，最后将这些功能模块组合，直到完成整个系统。这就如同搭积木，用小的模块不断组合，最后完成系统的设计。其设计方法示意图如图 6.1-1 所示。

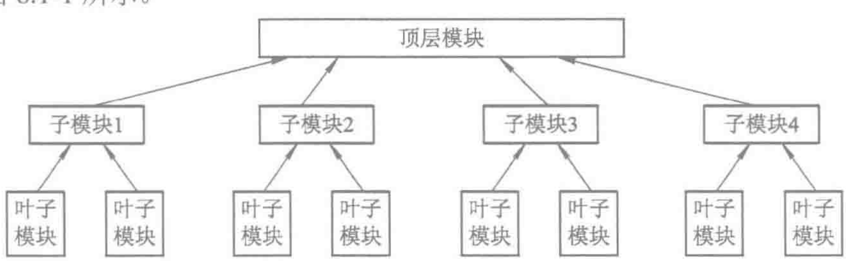


图 6.1-1 Bottom-Up 设计方法示意图

一个典型的 Bottom-Up 设计例子是在第 4 章中提到的串行加法器的设计。一个 4 位串行加法器由 4 个全加器构成，如图 6.1-2 所示。全加器是串行加法器的子模块，而全加器是由基本的逻辑门构成的，如图 6.1-3 所示，这些基本的逻辑门就是所说的叶子模块。这个设计中运用叶子模块(基本逻辑门)搭建成子模块(全加器)，再用子模块搭建成所需要的电路(串行加法器)。

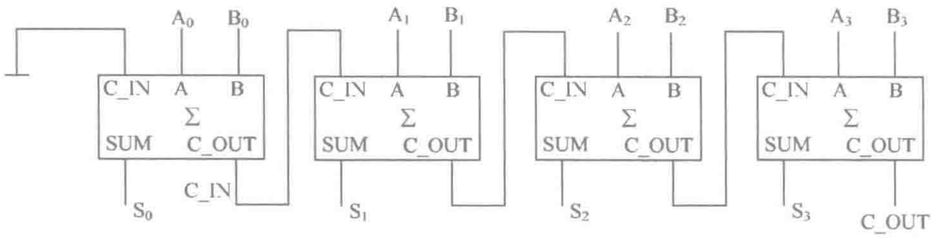


图 6.1-2 4 位串行加法器

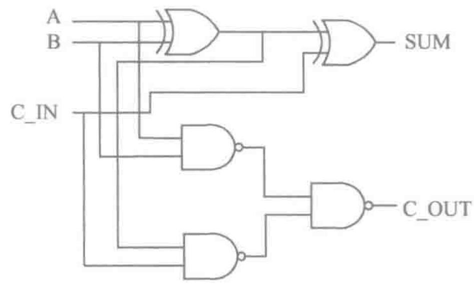


图 6.1-3 全加器逻辑电路

显然，Bottom-Up 设计方法没有明显的规律可循，主要依靠设计者的实践经验和熟练的设计技巧，用逐步试探的方法最后设计出一个完整的数字系统。系统的各项性能指标只有在系统构成后才能分析测试。

Bottom-Up 设计方法常用于原理图的设计中，相比于其它方法，该方法对于实现各个子模块电路所需的时间较短。但是该方法仍存在着许多不足之处，例如采用该设计方法容易对系统的整体功能把握不足，整个系统的设计周期比较长、效率低、设计质量难以保证等。因此，这种方法只适用于小规模电路的设计。

2. Top-Down 设计方法

随着电子技术的快速发展，传统的设计方法已经不能满足日益增长的系统要求，Top-Down 设计方法成为数字系统设计的主流设计方法。在这种设计方法中，首先从系统级入手，把系统划分为若干个子功能单元，并编制出相应的行为或结构模型；再将这些子功能单元进一步拆分，就这样不断地拆分，直到整个系统中各个模块的逻辑关系合理，便于逻辑电路级的设计和实现，这种最底层的功能模块被称为叶子模块。其设计方法示意图如图 6.1-4 所示。

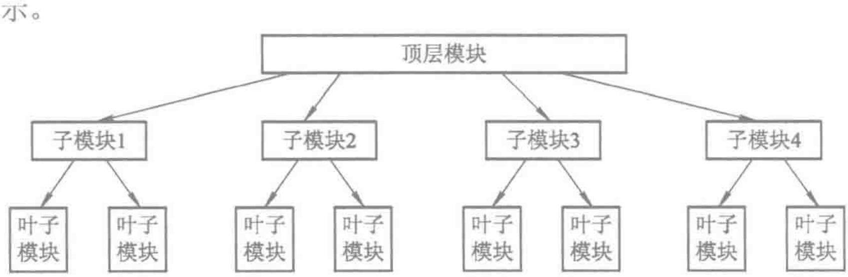


图 6.1-4 Top-Down 设计方法示意图

例如，使用 Top-Down 设计方法对一个典型 CPU 进行设计，如图 6.1-5 所示。根据 CPU 的功能将 CPU 分为控制单元、逻辑运算单元、存储单元。控制单元主要控制整个 CPU 的工

作，用于调控整个 CPU 按照指令执行相应的规定动作，控制单元又由指令控制器、时序控制器、总线控制器和中断控制器构成。逻辑运算单元可以分为算术逻辑运算单元 ALU 和浮点运算单元 FPU。存储单元可以分为通用寄存器和专用寄存器。这样通过功能的不断细化，将大模块划分成了更小的模块，当划分到指令控制器这一级时，就可以通过各种逻辑电路来实现了。

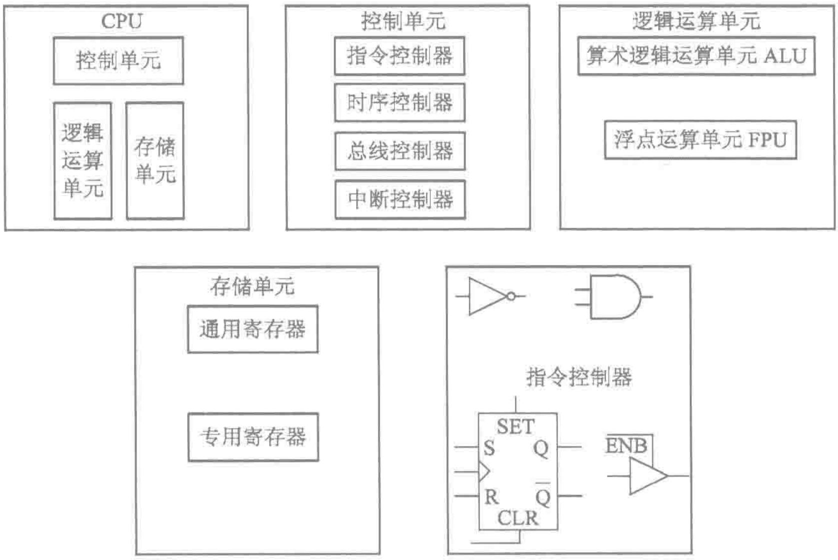


图 6.1-5 Top-Down 设计 CPU 流程

Top-Down 设计方法的优点是显而易见的，在整个设计过程中主要的仿真和调试过程是在高层次完成的，所以能够在设计的早期发现结构设计上的错误，及时进行调整，最大限度地不将错误带入到后续的设计环节中。同时方便了从系统进行划分和管理整个项目，使得大规模的复杂的数字电路实现成为可能，避免了不必要的重复设计，提高了设计效率。然而这种设计方法仍然存在着不足，即在设计的开始并不能准确地确定最终功能单元的设计，需要根据具体的设计情况，不断进行系统设计的修正。

现在的数字电路设计越来越复杂，单一的设计方法往往很难满足设计要求，因此通常将这两种设计方法结合起来，即采用混合的设计方法来进行设计。这样可以综合每种方法的优点，在高层系统设计时采用 Top-Down 设计方法，便于系统的划分；而在底层的设计时采用 Botton-Up 设计方法，这样可以缩短各个模块的设计时间。

下面以向量点积乘法器的设计为例，具体说明如何通过模块层次化的设计方法来完成电路的设计。

例 6.1-1 采用模块层次化设计方法，设计 4 维向量点积乘法器，其中向量 $\mathbf{a} = (a_1，a_2，a_3，a_{44})$，$\mathbf{b} = (b_1，b_2，b_3，b_4)$。点积乘法的规则为

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 \tag{6.1-1}$$

向量点积乘法是将向量的对应位置的值相乘，然后相加。其对应的电路结构如图 6.1-6 所示。

可以看出，4 位点积向量乘法器由 4 个乘法器和 3 个加法器构成，而加法器和乘法器又由半加器和全加器构成，其中半加器和全加器又由基本的逻辑门组成，其构成关系如图 6.1-7 所示。

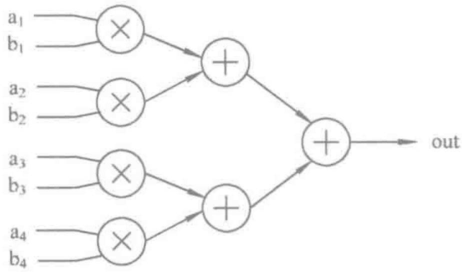


图 6.1-6　向量点积乘法器电路结构

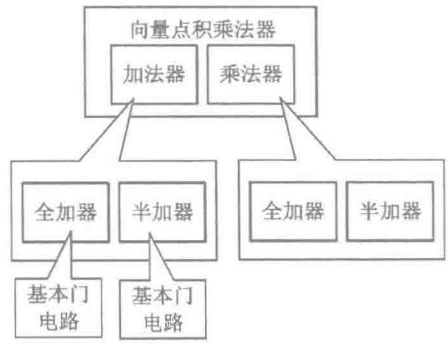


图 6.1-7　向量点积乘法器层次化功能模块

具体的 Verilog HDL 设计包括顶层模块 vector、加法器模块 add 和乘法器模块 mul_addtree。其 Verilog HDL 程序代码如下：

```
module vector(a1, a2, a3, a4, b1, b2, b3, b4, out);
    input [3:0] a1, a2, a3, a4, b1, b2, b3, b4;
    output [9:0] out;
    wire   [7:0] out1, out2, out3, out4;
    wire   [8:0] out5, out6;
    wire   [9:0] out;
    mul_addtree U1(.x(a1), .y(b1), .out(out1));
    mul_addtree U2(.x(a2), .y(b2), .out(out2));
    mul_addtree U3(.x(a3), .y(b3), .out(out3));
    mul_addtree U4(.x(a4), .y(b4), .out(out4));
    add #(8) U5(.a(out1), .b(out2), .out(out5));
    add #(8) U6(.a(out3), .b(out4), .out(out6));
    add #(9) U7(.a(out5), .b(out6), .out(out));
endmodule
//加法器
module add(a, b, out);
    parameter size = 8;
    input   [size-1:0] a, b;
    output [size:0] out;
    assign out = a+b;
endmodule
//乘法器
module mul_addtree(mul_a, mul_b, mul_out);
    input   [3:0] mul_a, mul_b;                    // IO 端口声明
```

```
    output [7:0] mul_out;
    wire      [3:0] mul_out;                              //连线类型声明
    wire      [3:0] stored0, stored1, stored2, stored3;
    wire      [3:0] add01, add23;
    assign stored3 = mul_b[3]?{1'b0, mul_a, 3'b0}:8'b0;   //逻辑设计
    assign stored2 = mul_b[2]?{2'b0, mul_a, 2'b0}:8'b0;
    assign stored1 = mul_b[1]?{3'b0, mul_a, 1'b0}:8'b0;
    assign stored0 = mul_b[0]?{4'b0, mul_a}:8'b0;
    assign add01 = stored1+stored0;
    assign add23 = stored3+stored2;
    assign mul_out = add01+add23;
endmodule
```

可以看到，在这个电路设计中，vector、add 和 mul_addtree 三个模块共同构成了向量点积乘法器。顶层模块 vector 采用结构描述方式分别使用了 4 个 mul_addtree 和 3 个 add 模块，分两层实现了电路功能。

6.2 典型电路设计

6.2.1 加法器树乘法器

加法器树乘法器的设计思想是“移位后加”，并且加法运算采用加法器树的形式。乘法运算的过程是，被乘数与乘数的每一位相乘并且乘以相应的权值，最后将所得的结果相加，便得到了最终的乘法结果。加法器树乘法器在计算被乘数与乘数及与其权值相乘时采用“判断移位”的形式，即通过判断乘数位是否为 1 来决定结果为 0 或者进行移位。当乘数位为 1 时被乘数根据该位的权值将被乘数移位，权值为 0 不移位，权值为 2 左移 1 位，权值为 4 左移 2 位，权值为 8 左移 3 位……；当乘数位为 0 时直接输出 0。而加法器树乘法器的加法运算是通过加法器树来实现的，如图 6.2-1 所示。加法器树所需要的加法器的数目是操作数的位数减 1。

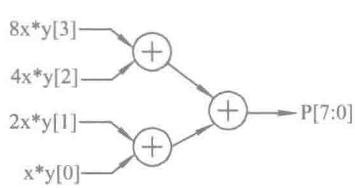


图 6.2-1 4 位的加法器树乘法器结构

例 6.2-1 图 6.2-1 是一个 4 位的加法器树乘法器结构，用 Verilog HDL 设计一个 4 位加法器树乘法器。

```
module mul_addtree(mul_a, mul_b, mul_out);
    input    [3:0] mul_a, mul_b;                          //IO 端口声明
```

```
    output [7:0] mul_out;

    wire      [7:0] mul_out;                              //连线类型声明
    wire      [7:0] stored0, stored1, stored2, stored3;
    wire      [7:0] add01, add23;

    assign stored3 = mul_b[3]?{1'b0, mul_a, 3'b0}:8'b0;      //逻辑设计
    assign stored2 = mul_b[2]?{2'b0, mul_a, 2'b0}:8'b0;
    assign stored1 = mul_b[1]?{3'b0, mul_a, 1'b0}:8'b0;
    assign stored0 = mul_b[0]?{4'b0, mul_a}:8'b0;
    assign add01 = stored1+stored0;
    assign add23 = stored3+stored2;
    assign mul_out = add01+add23;
endmodule
//****************************测试代码********************************
module mult_addtree_tb;
    reg [3:0]mult_a;
    reg [3:0]mult_b;
    wire [7:0]mult_out;
    //模块例化
    mul_addtree U1(.mul_a(mult_a), .mul_b(mult_b), .mul_out(mult_out));

    initial      //测试信号
        begin
            mult_a = 0;   mult_b = 0;
            repeat(9)
            begin
              #20 mult_a = mult_a+1;
            mult_b = mult_b+1;
        end
    end
endmodule
```

其测试结果如图 6.2-2 所示。

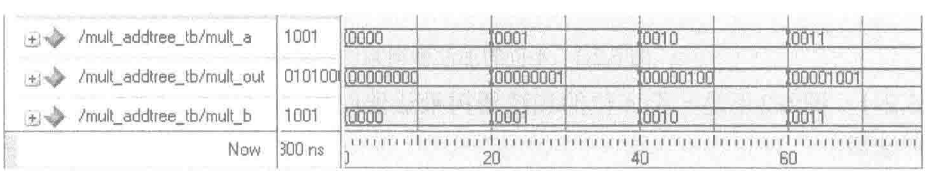


图 6.2-2　加法器树乘法器测试结果

加法器树乘法器可以通过流水线提高电路速度。其基本思想是加法器树中插入寄存器，将加法器树乘法器设计成流水线型。

例 6.2-2 用 Verilog HDL 设计一个两级流水线 4 位加法器树乘法器。

两级流水线 4 位加法器树乘法器的结构如图 6.2-3 所示。

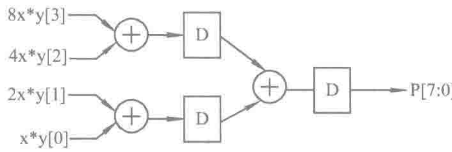


图 6.2-3 两级流水线 4 位加法器树乘法器的结构

通过在第一级与第二级、第二级与第三级加法器之间插入 D 触发器组，可以实现两级流水线设计，其 Verilog HDL 程序代码如下：

```
module mul_addtree_2_stage(clk, clr, mul_a, mul_b, mul_out);
    input   clk, clr;
    input   [3:0] mul_a, mul_b;                          // IO 端口声明
    output [7:0] mul_out;
    reg       [7:0] add_tmp_1, add_tmp_2, mul_out;
    wire      [7:0] stored0, stored1, stored2, stored3;
    assign stored3   =mul_b[3]?{1'b0, mul_a, 3'b0}:8'b0;          //逻辑设计
    assign stored2 = mul_b[2]?{2'b0, mul_a, 2'b0}:8'b0;
    assign stored1 = mul_b[1]?{3'b0, mul_a, 1'b0}:8'b0;
    assign stored0 = mul_b[0]?{4'b0, mul_a}:8'b0;

    always@(posedge clk or negedge clr)                  //时序控制
    begin
          if(!clr)
          begin
             add_tmp_1 <= 8'b0000_0000;
             add_tmp_2 <= 8'b0000_0000;
             mul_out <= 8'b0000_0000;
          end
    else
    begin
      add_tmp_1 <= stored3+stored2;
      add_tmp_2 <= stored1+stored0;
      mul_out <= add_tmp_1+add_tmp_2;
    end
  end
endmodule
```

应该指出，在这个 Verilog HDL 代码中，使用非阻塞赋值方式可以直接实现寄存器的插入，相关内容参照 3.2.3 节。

```
//****************************测试代码*****************************//
module mult_addtree_2_stag_tb;
    reg clk, clr;
    reg [3:0]mult_a, mult_b;
    wire [7:0]mult_out;
    mul_addtree_2_stage     U1(.mul_a(mult_a), .mul_b(mult_b),
                               .mul_out(mult_out), .clk(clk), .clr(clr));
      initial
      begin
          clk = 0; clr = 0; mult_a = 1; mult_b = 1;
          #5 clr = 1;
      end
    always #10 clk = ~clk;
    initial
      begin
          repeat(5)
          begin
            #20 mult_a = mult_a+1;    mult_b = mult_b+1;
          end
      end
endmodule
```

其测试结果如图 6.2-4 所示。

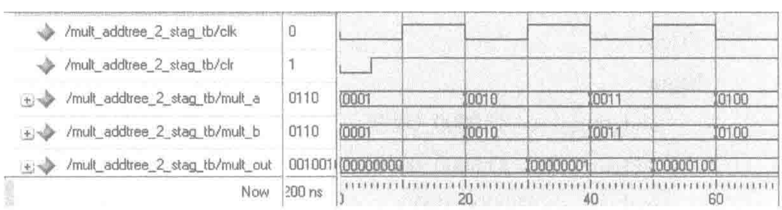


图 6.2-4　两级流水线 4 位加法器树乘法器的测试结果

6.2.2　Wallace 树乘法器

在乘法器的设计中，采用树形乘法器可以减少关键路径和所需的加法器单元数目。Wallace 树乘法器就是其中一种。下面以一个 4 × 4 位的乘法器为例来介绍 Wallace 树乘法器。Wallace 树乘法器原理图如图 6.2-5 所示，其中 FA 为全加器，HA 为半加器。其基本原理是，加法从数据最密集的地方开始，不断地反复使用全加器、半加器来覆盖“树”。这一级全加器是一个 3 输入 2 输出的器件，因此全加器又称为 3-2 压缩器。通过全加器将树的深度不断缩减，最终缩减为一个深度为 2 的树。最后一级则采用一个简单的 2 输入加法器组成。

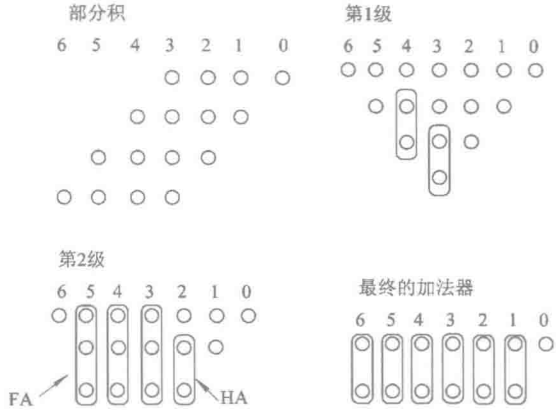


图 6.2-5　Wallace 树乘法器原理图

其电路结构如图 6.2-6 所示。

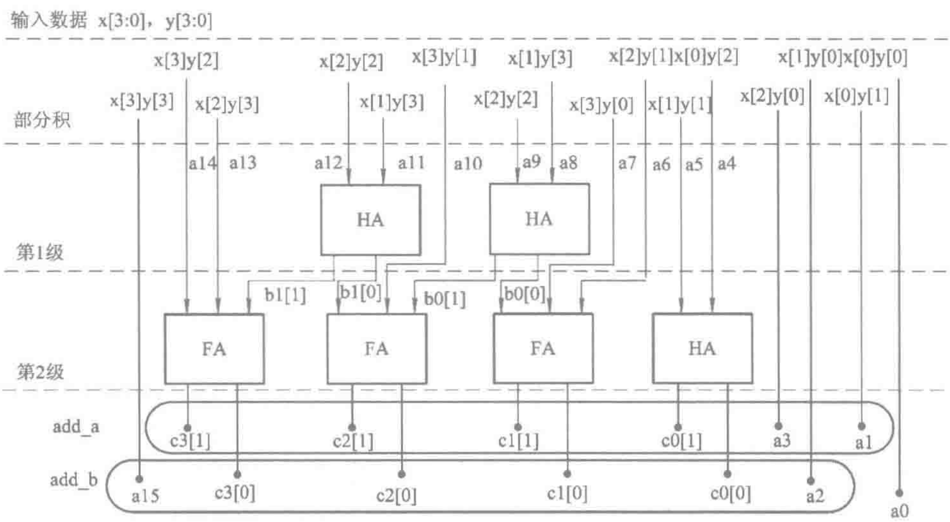


图 6.2-6　Wallace 树乘法器电路结构

例 6.2-3　用 Verilog HDL 设计如图 6.2-6 所示的 4 位 Wallace 树乘法器。

```
module wallace(x, y, out);
    parameter size=4;                    //参数定义
    input [size-1:0] x, y;
    output [2*size-1:0] out;             //端口声明
    wire   [size*size-1:0] a;
    wire   [1:0] b0, b1, c0, c1, c2, c3;  //连线类型声明
    wire   [5:0] add_a, add_b;
    wire   [6:0] add_out;
    wire   [2*size-1:0] out;
    assign
    a = {x[3], x[3], x[2], x[2], x[1], x[3], x[1], x[0], x[3], x[2], x[1], x[0], x[2], x[1], x[0], x[0]}&
    {y[3], y[2], y[3], y[2], y[3], y[1], y[2], y[3], y[0], y[1], y[1], y[2], y[0], y[0], y[1], y[0]}; //部分积
```

```
        hadd U1(.x(a[8]), .y(a[9]), .out(b0));                  //2 输入半加器
        hadd U2(.x(a[11]), .y(a[12]), .out(b1));
        hadd U3(.x(a[4]), .y(a[5]), .out(c0));
        fadd U4(.x(a[6]), .y(a[7]), .z(b0[0]), .out(c1));        //3 输入全加器
        fadd U5(.x(a[13]), .y(a[14]), .z(b0[1]), .out(c2));
        fadd U6(.x(b1[0]), .y(a[10]), .z(b1[1]), .out(c3));
        assign add_a = {c3[1], c2[1], c1[1], c0[1], a[3], a[1]};      //加数
        assign add_b = {a[15], c3[0], c2[0], c1[0], c0[0], a[2]};
        assign add_out = add_a+add_b;
        assign out = {add_out, a[0]};
    endmodule
    module fadd(x, y, z, out);
        output [1:0]out;
        input x, y, z;
        assign out=x+y+z;
    endmodule
    module hadd(x, y, out);
        output [1:0]out;
        input x, y;
        assign out=x+y;
    endmodule
    //***************************测试代码*********************//
    module wallace_tb;
        reg [3:0] x, y;
        wire [7:0] out;
      wallace m(.x(x), .y(y), .out(out));          //模块例化
        initial                                  //测试信号
            begin
                x = 3;   y = 4;
                #20 x = 2;   y = 3;
                #20 x = 6;   y = 8;
            end
    endmodule
```

其测试结果如图 6.2-7 所示。

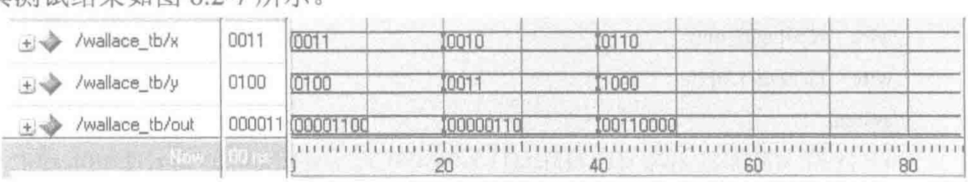


图 6.2-7　Wallace 树乘法器的测试结果

6.2.3 复数乘法器

复数乘法的算法是：设复数 $x = a + bi$， $y = c + di$，则复数乘法结果为

$$x \times y = (a + bi)(c + di) = (ac - bd) + i(ad + bc) \tag{6.2-1}$$

复数乘法器的电路结构如图 6.2-8 所示。将复数 x 的实部与复数 y 的实部相乘，减去 x 的虚部与 y 的虚部相乘，得到输出结果的实部。将 x 的实部与 y 的虚部相乘，加上 x 的虚部与 y 的实部相乘，得到输出结果的虚部。

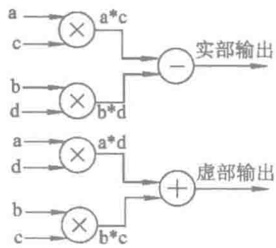


图 6.2-8 复数乘法器的电路结构

例 6.2-4 用 Verilog HDL 设计实部和虚部均为 4 位二进制数的复数乘法器。

```
module complex(a, b, c, d, out_real, out_im);
    input [3:0]a, b, c, d;
    output [8:0] out_real, out_im;
    wire [7:0] sub1, sub2, add1, add2;
    wallace U1(.x(a), .y(c), .out(sub1));
    wallace      U2(.x(b), .y(d), .out(sub2));
    wallace      U3(.x(a), .y(d), .out(add1));
    wallace      U4(.x(b), .y(c), .out(add2));
    assign       out_real = sub1-sub2;
    assign       out_im = add1+add2;
endmodule
//************************测试模块*********************//
module complex_tb;
    reg [3:0] a, b, c, d;
    wire [8:0] out_real;
    wire [8:0] out_im;
    complex U1(.a(a), .b(b), .c(c), .d(d), .out_real(out_real), .out_im(out_im));
    initial
        begin
            a = 2;      b = 2;   c = 5;   d = 4;
            #10
            a = 4;   b = 3;   c = 2;   d = 1;
            #10
```

```
            a = 3;   b = 2;   c = 3;   d = 4;
        end
    endmodule
```

其测试结果如图 6.2-9 所示。

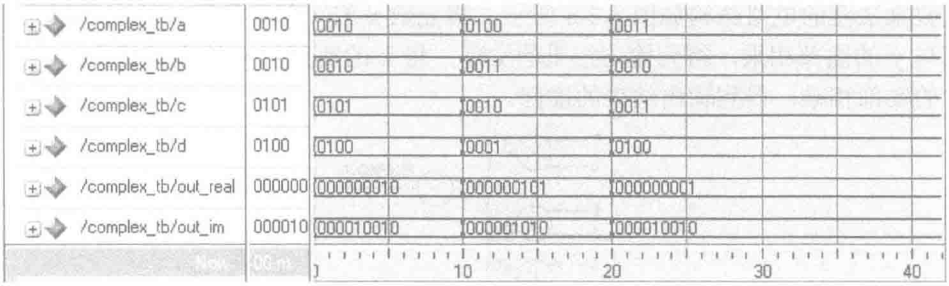


图 6.2-9 复数乘法器的测试结果

6.2.4 FIR 滤波器的设计

滤波器就是对特定的频率或者特定频率以外的频率进行消除的电路，被广泛用于通信系统和信号处理系统中。从功能角度而言，数字滤波器对输入离散信号的数字代码进行运算处理，以达到滤除频带外信号的目的。

有限冲激响应(FIR)滤波器就是一种常用的数字滤波器，采用对已输入样值的加权和来形成它的输出。其系统函数为

$$H(z) = \frac{y(z)}{x(z)} = a + bz^{-1} + cz^{-2} \tag{6.2-2}$$

其中，z^{-1} 表示延迟一个时钟周期，z^{-2} 表示延迟两个时钟周期。

对于输入序列 X[n]的 FIR 滤波器，可用图 6.2-10 所示的结构示意图来表示，其中 X[n]是输入数据流。各级的输入连接和输出连接被称为抽头，系数(b_0，b_1，…，b_n)被称为抽头系数。一个 M 阶的 FIR 滤波器将会有 M + 1 个抽头。通过移位寄存器用每个时钟边沿 n(时间下标)处的数据流采样值乘以抽头系数，并将它们加起来形成输出 Y[n]。

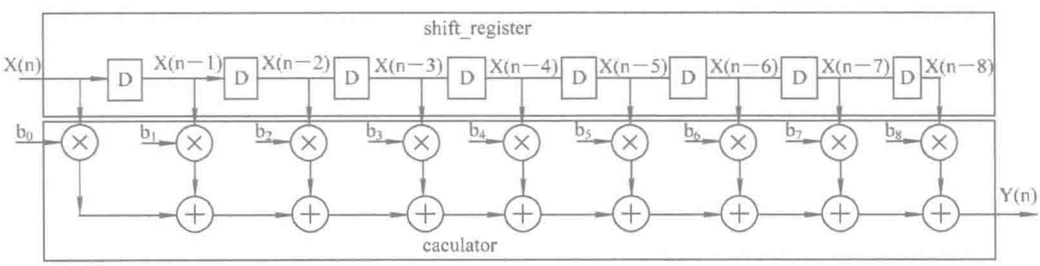


图 6.2-10 FIR 滤波器结构示意图

例 6.2-5 用 Verilog HDL 设计一个输入信号位宽为 4 bit 的 8 阶 FIR 滤波器。

如图 6.2-10 所示，FIR 滤波器电路有两个主要功能模块：移位寄存器组模块 shift_register 用于存储串行进入滤波器的数据；乘加计算模块 caculator 用于进行 FIR 计算(见式(6.2-2))。因此，在顶层模块中采用结构性的描述方式设计。

```
module FIR (Data_out, Data_in, clock, reset);
    output      [9:0]    Data_out;
    input       [3:0]    Data_in;
    input                clock, reset;
    wire        [9:0]    Data_out;
    wire        [3:0]    samples_0, samples_1, samples_2, samples_3, samples_4,
                         samples_5, samples_6, samples_7, samples_8;
    shift_register U1(.Data_in(Data_in), .clock(clock), .reset(reset),
                      .samples_0(samples_0), .samples_1(samples_1),
                      .samples_2(samples_2), .samples_3(samples_3),
                      .samples_4(samples_4), .samples_5(samples_5),
                      .samples_6(samples_6), .samples_7(samples_7),
                      .samples_8(samples_8));
    caculator U2(.samples_0(samples_0), .samples_1(samples_1),
                      .samples_2(samples_2), .samples_3(samples_3),
                      .samples_4(samples_4), .samples_5(samples_5),
                      .samples_6(samples_6), .samples_7(samples_7),
                      .samples_8(samples_8), .Data_out(Data_out));
endmodule
```

shift_register 模块用于存储输入的数据流，本例中主要负责存储 8 个位宽为 4 bit 的输入数据信号，用于 caculator 模块的输入。

```
module shift_register(Data_in, clock, reset, samples_0, samples_1, samples_2,
                      samples_3, samples_4, samples_5, samples_6,
                      samples_7, samples_8);
    input   [3:0] Data_in;
    input   clock, reset;
    output [3:0] samples_0, samples_1, samples_2, samples_3, samples_4,
                 samples_5, samples_6, samples_7, samples_8;
    reg     [3:0] samples_0, samples_1, samples_2, samples_3, samples_4,
                 samples_5, samples_6, samples_7, samples_8;
    always@(posedge clock or negedge reset)
        begin
            if(reset)
              begin
                  samples_0 <= 4'b0;
                  samples_1 <= 4'b0;
                  samples_2 <= 4'b0;
                  samples_3 <= 4'b0;
                  samples_4 <= 4'b0;
```

```
                samples_5 <= 4'b0;
                samples_6 <= 4'b0;
                samples_7 <= 4'b0;
                samples_8 <= 4'b0;
            end
          else
            begin
                samples_0 <= Data_in;
                samples_1 <= samples_0;
                samples_2 <= samples_1;
                samples_3 <= samples_2;
                samples_4 <= samples_3;
                samples_5 <= samples_4;
                samples_6 <= samples_5;
                samples_7 <= samples_6;
                samples_8 <= samples_7;
            end
       end
endmodule
```

caculator 模块用于进行 8 输入信号与抽头系数的乘法和累加，并产生滤波之后的输出信号 Data_out。应该指出的是，FIR 滤波器系数具有对称性，在本例中，$b_0=b_8$，$b_1=b_7$，$b_2=b_6$，$b_3=b_5$，因此，可以采用先将输入信号相加再与抽头系数相乘的方式，减少乘法器电路的数量和芯片面积。

```
module caculator(samples_0, samples_1, samples_2, samples_3, samples_4, samples_5,
                 samples_6, samples_7, samples_8, Data_out);
    input [3:0] samples_0, samples_1, samples_2, samples_3, samples_4, samples_5,
                samples_6, samples_7, samples_8;
    output [9:0] Data_out;
    wire   [9:0] Data_out;
    wire   [3:0] out_tmp_1, out_tmp_2, out_tmp_3, out_tmp_4, out_tmp_5;
    wire   [7:0] out1, out2, out3, out4, out5;
        parameter  b0 = 4'b0010;
        parameter  b1 = 4'b0011;
        parameter  b2 = 4'b0110;
        parameter  b3 = 4'b1010;
        parameter  b4 = 4'b1100;
        mul_addtree U1(.mul_a(b0), .mul_b(out_tmp_1), .mul_out(out1));
        mul_addtree U2(.mul_a(b1), .mul_b(out_tmp_2), .mul_out(out2));
        mul_addtree U3(.mul_a(b2), .mul_b(out_tmp_3), .mul_out(out3));
```

```
        mul_addtree U4(.mul_a(b3), .mul_b(out_tmp_4), .mul_out(out4));
        mul_addtree U5(.mul_a(b4), .mul_b(samples_4), .mul_out(out5));
        assign out_tmp_1 = samples_0+samples_8;
        assign out_tmp_2 = samples_1+samples_7;
        assign out_tmp_3 = samples_2+samples_6;
        assign out_tmp_4 = samples_3+samples_5;
        assign Data_out = out1+out2+out3+out4+out5;
endmodule
```

乘法器模块 mul_addtree 的代码见例 6.2-2。

```
//**********************测试模块*********************//
module FIR_tb;
    reg clock, reset;
    reg [3:0] Data_in;
    wire [9:0] Data_out;
    FIR     U1 (.Data_out(Data_out), .Data_in(Data_in),
                .clock(clock), .reset(reset));
    initial
      begin
            Data_in = 0;     clock = 0;     reset = 1;
            #10 reset = 0;
      end
    always
      begin
            #5 clock <= ~clock;
            #5 Data_in <= Data_in+1;
      end
endmodule
```

其测试结果如图 6.2-11 所示。

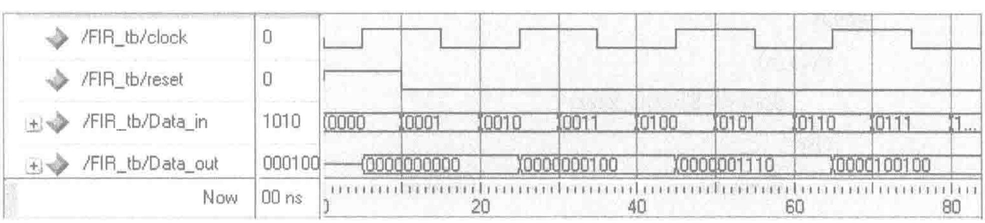


图 6.2-11　FIR 滤波器的测试结果

6.2.5　片内存储器的设计

片内存储器分为 RAM 和 ROM 两大类。本节将分别介绍 RAM 和 ROM 的 Verilog HDL 描述方式。

1. RAM 的 Verilog HDL 描述

RAM 是随机存储器，存储单元的内容可按需随意取出或存入。这种存储器在断电后将丢失所有数据，因此一般用来存储一些短时间内使用的程序和数据。其内部结构如图 6.2-12 所示。

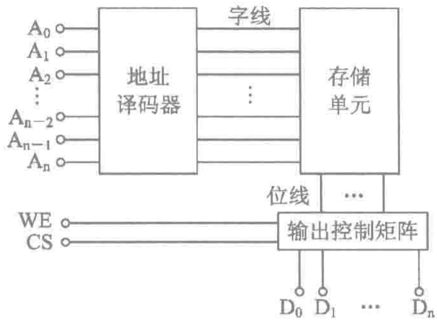


图 6.2-12　RAM 的内部结构

根据地址总线、数据总线以及读写控制线的数目，可以将 RAM 分为单端口 RAM 和双端口 RAM 两大类。

例 6.2-6　用 Verilog HDL 设计深度为 8、位宽为 8 bit 的单端口 RAM。

单端口 RAM 只有一套地址总线，读操作和写操作是分开的。其 Verilog HDL 程序代码如下：

```
module ram_single(clk, addm, cs_n, we_n, din, dout);
    input           clk;            //时钟信号
    input [2:0]     addm;           //地址信号
    input           cs_n;           //片选信号
    input           we_n;           //写使能信号
    input [7:0] din;                //输入数据
    output[7:0] dout;               //输出数据
    reg     [7:0] dout;
    reg     [7:0] raml [7:0];       //8*8 bit 寄存器
    always@(posedge clk)
        begin
          if(cs_n)
              dout <= 8'bzzzz_zzzz;
          else
              if(we_n)                      //读数据
                dout <= raml[addm];
              else                          //写数据
                raml[addm] <= din;
    end
endmodule
//*******************************测试程序*******************************
```

```
module ram_single_tb;
    reg clk, we_n, cs_n;
    reg [2:0]addm;
    reg [7:0]din;
    wire [7:0]dout;
    ram_single U1(.clk(clk), .addm(addm), .cs_n(cs_n), .we_n(we_n),
                  .din(din), .dout(dout));
    initial
      begin
          clk = 0;   addm = 0;   cs_n = 1;   we_n = 0;   din = 0;
          #5 cs_n = 0;
          #315 we_n = 1;
      end
    always #10 clk = ~clk;
    initial
       begin
              repeat(7)
                  begin
                      #40 addm = addm+1;
                      din = din+1;
                  end
              #40
                  repeat(7)
                      #40 addm = addm-1;
       end
endmodule
```

其测试结果如图 6.2-13 所示。

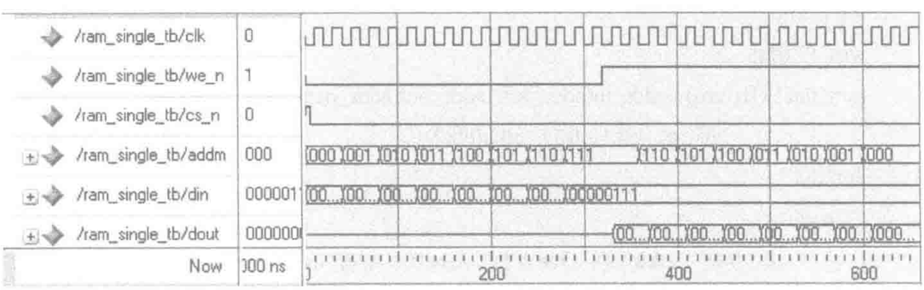


图 6.2-13　单端口 RAM 的测试结果

例 6.2-7　用 Verilog HDL 设计深度为 8、位宽为 8 bit 的双端口 RAM。

双端口 RAM 具有两套地址总线，一套用于读数据，另一套用于写数据。二者可以分别进行操作。其 Verileg HDL 程序代码如下：

```
module ram_dual(q, addr_in, addr_out, d, we, rd, clk1, clk2);
    output [7:0] q;              //输出数据
    input  [7:0] d;              //输入数据
    input  [2:0] addr_in;        //写数据地址信号
    input  [2:0] addr_out;       //输出数据地址信号
    input  we;                   //写数据控制信号
    input  rd;                   //读数据控制信号
    input  clk1;                 //写数据时钟信号
    input  clk2;                 //读数据时钟信号
    reg    [7:0] q;
    reg    [7:0] mem[7:0];       //8*8 bit 寄存器
    always@(posedge clk1)
      begin
        if(we)
          mem[addr_in] <= d;
      end
    always@(posedge clk2)
      begin
        if(rd)
          q <= mem[addr_out];
      end
endmodule
//********************************测试代码*****************************
module ram_dual_tb;
    reg clk1, clk2, we, rd;
    reg [2:0]addr_in;
    reg [2:0]addr_out;
    reg [7:0]d;
    wire [7:0]q;
    ram_dual U1(.q(q), .addr_in(addr_in), .addr_out(addr_out), .d(d), .we(we),
                .rd(rd), .clk1(clk1), .clk2(clk2));
    initial
      begin
        clk1 = 0;   clk2 = 0;   we = 1;   rd = 0;   addr_in = 0; addr_out = 0; d = 0;
        #320 we = 0;
        rd = 1;
      end
    always
      begin
```

```
            #10 clk1 = ~clk1;   clk2 = ~clk2;
        end
    initial
        begin
            repeat(7)
            begin
                    #40 addr_in = addr_in+1;   d = d+1;
            end
                    #40
                    repeat(7)   #40 addr_out = addr_out+1;
        end
endmodule
```

其测试结果如图 6.2-14 所示。

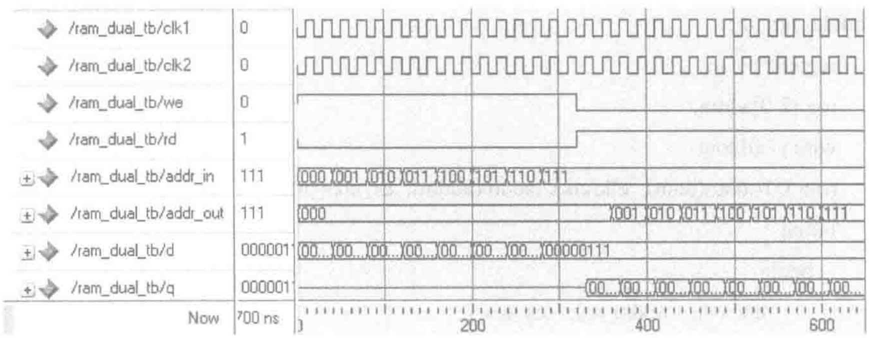


图 6.2-14 双端口 RAM 的测试结果

2. ROM 的 Verilog HDL 描述

ROM 即只读存储器，是一种只能读出事先存储的数据的存储器，其特性是存入的数据无法改变。也就是说，这种存储器只能读不能写。由于 ROM 在断电之后数据不会丢失，所以通常用在不需经常变更资料的电子或电脑系统中，资料并不会因为电源的关闭而消失。

例 6.2-8 用 Verilog HDL 设计深度为 8、位宽为 8 bit 的 ROM。

```
module rom(dout, clk, addm, cs_n);
    input   clk, cs_n;
    input   [2:0] addm;
    output [7:0] dout;
    reg      [7:0] dout;
    reg      [7:0] rom[7:0];
    initial
      begin
            rom[0] = 8'b0000_0000;
            rom[1] = 8'b0000_0001;
            rom[2] = 8'b0000_0010;
```

```
            rom[3] = 8'b0000_0011;
            rom[4] = 8'b0000_0100;
            rom[5] = 8'b0000_0101;
            rom[6] = 8'b0000_0110;
            rom[7] = 8'b0000_0111;
        end
          always@(posedge clk)
              begin
                    if(cs_n)      dout <= 8'bzzzz_zzzz;
                    else          dout <= rom[addm];
              end
endmodule
//******************************测试代码****************************
module rom_tb;
    reg clk, cs_n;
    reg [2:0]addm;
    wire [7:0]dout;
    rom U1(.dout(dout), .clk(clk), .addm(addm), .cs_n(cs_n));
    initial
      begin
          clk = 0;   addm = 0;   cs_n = 0;
      end
    always #10 clk = ~clk;
    initial
      begin
      repeat(7)
      #20 addm = addm+1;
    end
endmodule
```

其测试结果如图 6.2-15 所示。

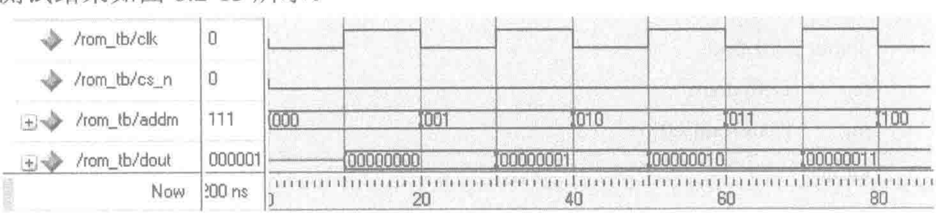


图 6.2-15　ROM 的测试结果

6.2.6　FIFO 设计

FIFO(First In First Out)是一种先进先出的数据缓存器，通常用于接口电路的数据缓存。

它与普通存储器的区别是，没有外部读写地址线，可以使用两个时钟分别进行写操作和读操作。FIFO 只能顺序写入数据和顺序读出数据，其数据地址由内部读写指针自动加 1 完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。

FIFO 由存储器块和对数据进出 FIFO 的通道进行管理的控制器构成，每次只对一个寄存器提供存取操作，而不是对整个寄存器阵列进行。FIFO 有两个地址指针，一个用于将数据写入下一个可用的存储单元，一个用于读取下一个未读存储单元的操作。其读写数据必须一次进行。其读写过程如图 6.2-16 所示。

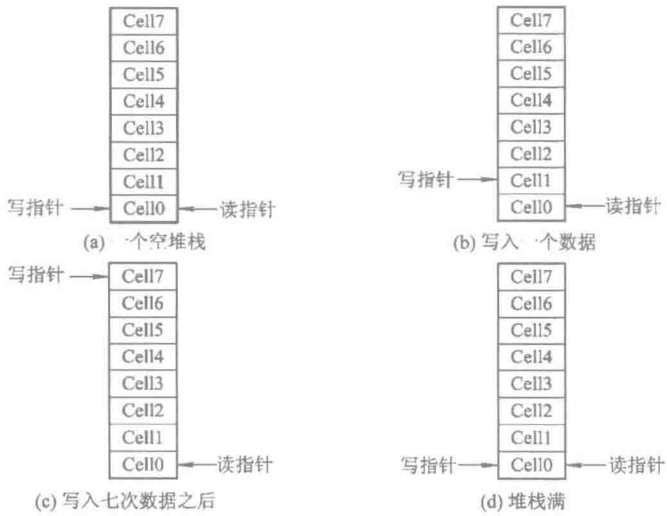


图 6.2-16　FIFO 的读写过程

当一个堆栈为空时(图 6.2-16(a))，读数据指针和写数据指针都指向第一个存储单元；当写入一个数据时(图 6.2-16(b))，写数据指针将指向下一个存储单元；经过七次写数据操作后(图 6.2-16(c))，写指针将指向最后一个存储单元；当经过连续八次写操作之后，写指针将回到首单元并且显示堆栈状态为满(图 6.2-16(d))。数据的读操作和写操作相似，当读出一个数据时，读数据指针将移向下一个存储单元，直到读出全部的数据，此时读指针回到首单元，堆栈状态显示为空。

一个 FIFO 的组成一般包括两部分：地址控制部分和存储数据的 RAM 部分，如图 6.2-17 所示。地址控制部分可以根据读写指令生成 RAM 地址。RAM 用于存储堆栈数据，并根据控制部分生成的地址信号进行数据的存储和读取操作。这里的 RAM 采用的是前面提到的双端口 RAM。

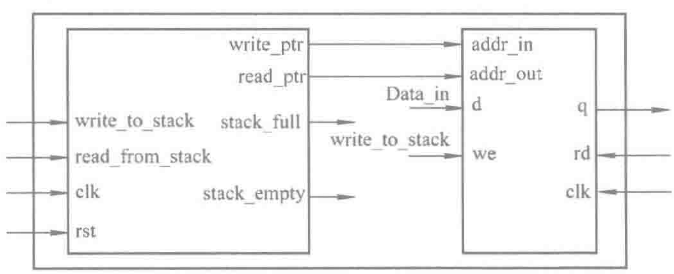


图 6.2-17　FIFO 结构图

例 6.2-9 用 Verilog HDL 设计深度为 8、位宽为 8 bit 的 FIFO。

```
//顶层模块
module FIFO_buffer(clk, rst, write_to_stack, read_from_stack, Data_in, Data_out);
    input   clk, rst;
    input   write_to_stack, read_from_stack;
    input   [7:0] Data_in;
    output [7:0] Data_out;
    wire [7:0]Data_out;
    wire stack_full, stack_empty;
    wire [2:0] addr_in, addr_out;
    FIFO_control U1(.stack_full(stack_full), .stack_empty(stack_empty),
                    .write_to_stack(write_to_stack), .write_ptr(addr_in),
                    .read_ptr(addr_out), .read_from_stack(read_from_stack),
                    .clk(clk), .rst(rst));
    ram_dual      U2(.q(Data_out), .addr_in(addr_in), .addr_out(addr_out),
                     .d(Data_in), .we(write_to_stack), .rd(read_from_stack),
                     .clk1(clk), .clk2(clk));
endmodule
//控制模块
module FIFO_control(write_ptr, read_ptr, stack_full, stack_empty, write_to_stack,
                     read_from_stack, clk, rst);

     parameter stack_width = 8;
parameter stack_height = 8;
parameter stack_ptr_width = 3;
output    stack_full;                              //堆栈满标志
output    stack_empty;                             //堆栈空标志
output    [stack_ptr_width-1:0] read_ptr;          //读数据地址
output    [stack_ptr_width-1:0] write_ptr;         //写数据地址
input     write_to_stack;                          //将数据写入堆栈
input     read_from_stack;                         //从堆栈读出数据
input     clk;
input     rst;
reg       [stack_ptr_width-1:0] read_ptr;
reg       [stack_ptr_width-1:0] write_ptr;
reg       [stack_ptr_width:0]   ptr_gap;
reg       [stack_width-1:0]      Data_out;
reg       [stack_width-1:0]      stack[stack_height-1:0];
//堆栈状态信号
```

```
assign stack_full = (ptr_gap == stack_height);
assign stack_empty = (ptr_gap == 0);
always@(posedge clk or posedge rst)
    begin
        if(rst)
            begin
                Data_out <= 0;
                read_ptr <= 0;
                write_ptr <= 0;
                ptr_gap <= 0;
            end
        else if(write_to_stack && (!stack_full) && (!read_from_stack))
            begin
                write_ptr <= write_ptr+1;
                ptr_gap <= ptr_gap+1;
            end
           else if(!write_to_stack && (!stack_empty) && (read_from_stack))
            begin
                read_ptr <= read_ptr+1;
                ptr_gap <= ptr_gap-1;
            end
else if(write_to_stack && stack_empty && read_from_stack)
            begin
                write_ptr <= write_ptr+1;
                ptr_gap <= ptr_gap+1;
            end
else if(write_to_stack && stack_full && read_from_stack)
            begin
                read_ptr <= read_ptr+1;
                ptr_gap <= ptr_gap-1;
            end
else if(write_to_stack && read_from_stack
            && (!stack_full)&&(!stack_empty))
      begin
            read_ptr <= read_ptr+1;
            write_ptr <= write_ptr+1;
      end
end
endmodule
```

RAM 模块可参考例 6.2-7。

```
//******************************测试代码***************************
module FIFO_tb;
    reg clk, rst;
    reg [7:0]Data_in;
    reg write_to_stack, read_from_stack;
    wire [7:0] Data_out;
      FIFO_buffer     U1(.clk(clk), .rst(rst), .write_to_stack(write_to_stack),
                         .read_from_stack(read_from_stack), .Data_in(Data_in),
                         .Data_out(Data_out));

    initial
      begin
          clk = 0;   rst = 1;   Data_in = 0;   write_to_stack = 1;   read_from_stack = 0;
          #5 rst = 0;
          #155 write_to_stack = 0;
                read_from_stack = 1;
      end
      always #10 clk = ~clk;
    initial
      begin
          repeat(7)
          #20 Data_in = Data_in+1;
      end
endmodule
```

其测试结果如图 6.2-18 所示。

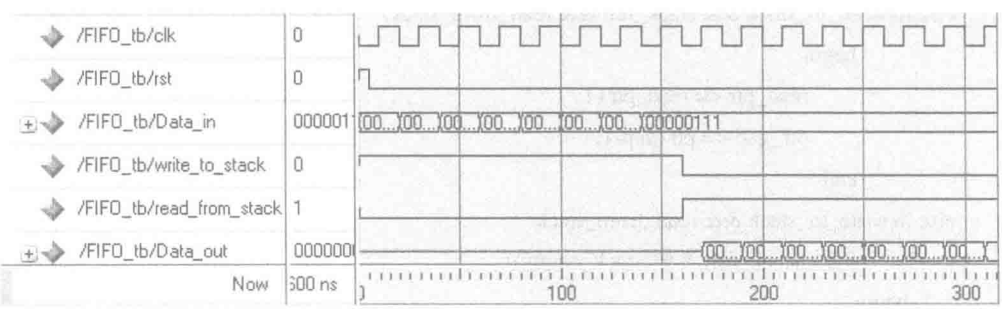


图 6.2-18　FIFO 的测试结果

6.2.7　键盘扫描和编码器

键盘扫描和编码器用于在拥有键盘的数字系统中手工输入数据，通过检测按键是否按

下来产生一个唯一对应此按键的扫描码。

例 6.2-10　用 Verilog HDL 设计一个十六进制键盘电路的键盘扫描和编码器。

图 6.2-19 是十六进制键盘电路的键盘扫描和编码器的一种设计方案。

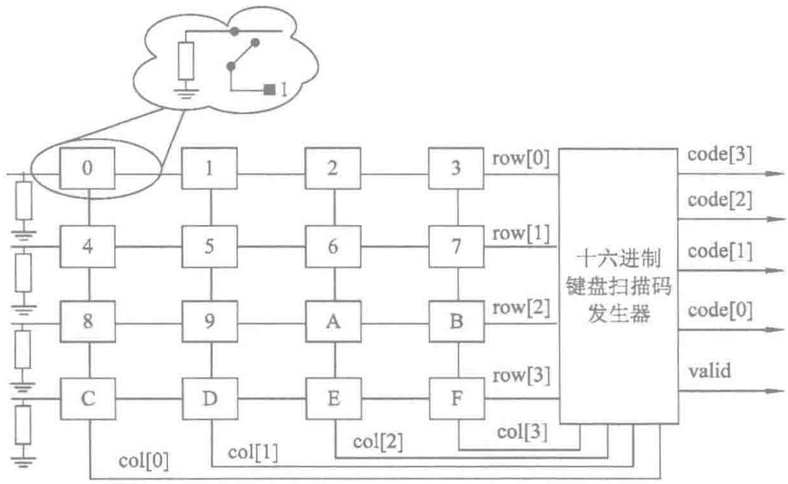


图 6.2-19　十六进制键盘电路的键盘扫描和编码器

键盘的每一行都会通过一个下拉电阻连接到地，而行和列之间有一个以按键为开关的连接，当按下按键时行和列之间就建立了连接。如果列线的值为 1，则由按键连接到的那个列的行值也将会被拉升为 1，反之行线将会被下拉至 0。通过键盘扫描码发生器单元来控制列线，使列线上的电压有规律而有效地检测行线的电压值，进而来确定按下按键的位置。

键盘扫描码发生器必须具备检测识别按键，并产生一个与按键唯一对应扫描码输出的功能。为此可以设计一个时序状态机，通过不同状态之间的转换来实现信号的检测识别以及对应扫描码的输出。首先，令所有的列线同时为 1 来检测行线的值，直到发现一条行线的值被上拉为 1，此时说明有按键被按下，但是仍然不能确定按键的位置；然后，通过依次将列线的值拉高来检测各个行线的值，直到检测到一条行线的值为 1 为止。其控制信号状态的转移图如图 6.2-20 所示。

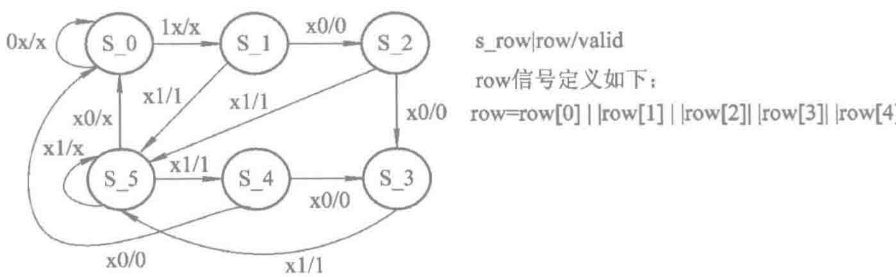


图 6.2-20　十六进制键盘电路的键盘扫描和编码器的控制信号状态转移图

此时行线和列线的交叉处就是按键的位置。根据已确定的按键的位置输出其对应的编码信息。其键盘编码表如表 6.2-1 所示。

表 6.2-1　十六进制键盘电路的编码器码表

key	row[3:0]	col[3:0]	code
0	0001	0001	0000
1	0001	0010	0001
2	0001	0100	0010
3	0001	1000	0011
4	0010	0001	0100
5	0010	0010	0101
6	0010	0100	0110
7	0010	1000	0111
8	0100	0001	1000
9	0100	0010	1001
A	0100	0100	1010
B	0100	1000	1011
C	1000	0001	1100
D	1000	0010	1101
E	1000	0100	1110
F	1000	1000	1111

为了使测试更接近于真实的物理环境，测试平台中必须包括模拟按键状态的信号发生器，能确认按键对应行线的模块 row_signal 和被测试模块 hex_keypad_grayhill_072。模拟按键状态的信号发生器可以嵌入在测试平台中，通过不断地给 key 信号赋值，模拟产生不同的按键信号。row_signal 模块用于检测按键的有效性并确定按键所处的行。而 synchronizer 模块通过检测各个行线值的或逻辑来确定是否有按键按下，当此模块的输出发生变化时，被测模块 hex_keypad_grayhill_072 将会确定按键的位置并输出相应的代码。其 Verilog HDL 程序代码如下：

```
//顶层模块
module keypad(clock, reset, row, code, valid, col);
    input    clock, reset;
    input    [3:0] row;
    output [3:0] code;
    output valid;
    output [3:0] col;
    wire s_row;
    hex_keypad_grayhill   U1(.code(code), .col(col), .valid(valid),
                            .row(row), .s_row(s_row), .clock(clock),
                            .reset(reset));
    synchronizer U2(.s_row(s_row), .row(row), .clock(clock), .reset(reset));
endmodule
```

```
//编码模块
module hex_keypad_grayhill(code, col, valid, row, s_row, clock, reset);
    output[3:0]   code;
    output        valid;
    output[3:0]   col;
    input[3:0]    row;
    input         s_row;
    input         clock, reset;
    reg[3:0]      col;
    reg[3:0]      code;
    reg [5:0]     state, next_state;
    parameter   s_0 = 6'b000001, s_1 = 6'b000010, s_2 = 6'b000100;
    parameter   s_3 = 6'b001000, s_4 = 6'b010000, s_5 = 6'b100000;
    assign valid = ((state == s_1) || (state == s_2) || (state == s_3) || (state == s_4))
            &&row;
    always@(row or col)
      case({row, col})
          8'b0001_0001: code = 0;        //0
          8'b0001_0010: code = 1;        //1
          8'b0001_0100: code = 2;        //2
          8'b0001_1000: code = 3;        //3
          8'b0010_0001: code = 4;        //4
          8'b0010_0010: code = 5;        //5
          8'b0010_0100: code = 6;        //6
          8'b0010_1000: code = 7;        //7
          8'b0100_0001: code = 8;        //8
          8'b0100_0010: code = 9;        //9
          8'b0100_0100: code = 10;       //A
          8'b0100_1000: code = 11;       //B
          8'b1000_0001: code = 12;       //C
          8'b1000_0010: code = 13;       //D
          8'b1000_0100: code = 14;       //E
          8'b1000_1000: code = 15;       //F
          default          code = 0;
       endcase
     always@(state or s_row or row)  //状态转化逻辑
       begin
           next_state = state;     col = 0;
           case(state)
```

```
                s_0:begin
                    col = 15;
                    if(s_row) n    ext_state = s_1;
                  end
                s_1:begin
                    col = 1;
                    if(row)        next_state = s_5;
                    else           next_state = s_2;
                  end
                s_2:begin
                    col = 2;
                    if(row)        next_state = s_5;
                    else           next_state = s_3;
                  end
                s_3:begin
                    col = 4;
                    if(row)        next_state = s_5;
                    else           next_state = s_4;
                  end
                s_4:begin
                    col = 8;
                    if(row)        next_state = s_5;
                    else           next_state = s_0;
                  end
                s_5:begin
                    col = 15;
                    if(!row)       next_state = s_0;
                  end
            endcase
        end
      always@(posedge clock or posedge reset)
        if(reset)        state <= s_0;
        else        state <= next_state;
endmodule
//**************************************************//
module synchronizer(s_row, row, clock, reset);
      output       s_row;
      input [3:0]   row;
      input         clock, reset;
```

```
    reg              a_row, s_row;
    always@(negedge clock or posedge reset)
        begin
            if(reset)
              begin
                  a_row <= 0;
                   s_row <= 0;
              end
            else
              begin
                  a_row <= (row[0]||row[1]||row[2]||row[3]);
                  s_row <= a_row;
              end
          end
endmodule
//****************************测试代码**************************
//模拟键盘产生信号
module row_signal(row, key, col);
    output   [3:0] row;
    input    [15:0] key;
    input    [3:0] col;
    reg     [3:0] row;
    always@(key or col)
      begin
          row[0] = key[0]&&col[0]||key[1]&&col[1]||key[2]&&col[2]||key[3]
          &&col[3];
          row[1] = key[4]&&col[0]||key[5]&&col[1]||key[6]&&col[2]||key[7]
          &&col[3];
          row[2] = key[8]&&col[0]||key[9]&&col[1]||key[10]&&col[2]||key[11]
          &&col[3];
          row[3] = key[12]&&col[0]||key[13]&&col[1]||key[14]&&col[2]||key[15]
          &&col[3];
      end
endmodule
//测试模块
module hex_keypad_grayhill_tb;
    wire [3:0] code;
    wire valid;
    wire [3:0] col;
```

```
wire [3:0] row;
reg clock;
reg reset;
reg   [15:0] key;
integer   j, k;
reg   [39:0] pressed;
parameter   [39:0] key_0 = "key_0";
parameter   [39:0] key_1 = "key_1";
parameter   [39:0] key_2 = "key_2";
parameter   [39:0] key_3 = "key_3";
parameter   [39:0] key_4 = "key_4";
parameter   [39:0] key_5 = "key_5";
parameter   [39:0] key_6 = "key_6";
parameter   [39:0] key_7 = "key_7";
parameter   [39:0] key_8 = "key_8";
parameter   [39:0] key_9 = "key_9";
parameter   [39:0] key_A = "key_A";
parameter   [39:0] key_B = "key_B";
parameter   [39:0] key_C = "key_C";
parameter   [39:0] key_D = "key_D";
parameter   [39:0] key_E = "key_E";
parameter   [39:0] key_F = "key_F";
parameter   [39:0] None = "None";
keypad U1(.clock(clock), .reset(reset), .row(row), .code(code), .vaild(vaild), .col(col));
//顶层模块
row_signal U2(.row(row), .key(key), .col(col));   //产生测试信号
always@(key)begin
    case(key)
        16'h0000: pressed = None;
        16'h0001: pressed = key_0;
        16'h0002: pressed = key_1;
        16'h0004: pressed = key_2;
        16'h0008: pressed = key_3;
        16'h0010: pressed = key_4;
        16'h0020: pressed = key_5;
        16'h0040: pressed = key_6;
        16'h0080: pressed = key_7;
        16'h0100: pressed = key_8;
        16'h0200: pressed = key_9;
```

```
            16'h0400: pressed = key_A;
            16'h0800: pressed = key_B;
            16'h1000: pressed = key_C;
            16'h2000: pressed = key_D;
            16'h4000: pressed = key_E;
            16'h8000: pressed = key_F;
            default:pressed = None;
      endcase
   end

   initial #2000 $stop;
   initial begin clock = 0;
                  forever #5 clock = ~clock;
         end
   initial begin reset = 1;
                  #10 reset = 0;
         end
   initial begin for(k = 0; k <= 1; k = k+1)
                  begin key = 0;
                    #20 for(j = 0; j <= 16; j = j+1)
                      begin
                         #20 key[j] = 1;
                         #60 key = 0;
                      end
                   end
          end
endmodule
```

其测试结果如图 6.2-21 所示。

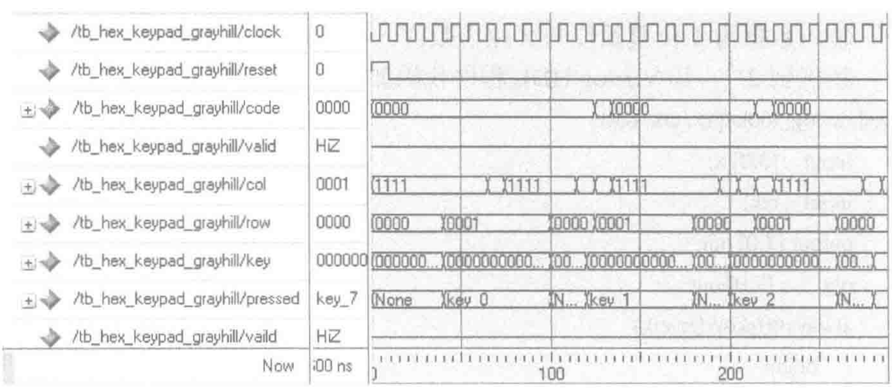


图 6.2-21　十六进制键盘电路的键盘扫描和编码器的测试结果

6.2.8 log 函数的 Verilog HDL 设计

ASIC 和 FPGA 的一个重要功能是实现计算函数加速器，随着通信、自动控制和多媒体信号处理计算量的增大，采用 Verilog HDL 设计计算函数加速器越来越重要。本小节和 6.2.9 小节将分别对两种典型的计算函数加速器进行设计。通过这些例子，可了解一些初步的设计方法。

log 函数是一种典型的单目计算函数，相应的还有指数函数、三角函数等。对于单目计算函数的硬件加速器的设计，一般有两种简单的方法：一种是查找表的方式；一种是使用泰勒级数展开成多项式进行近似计算。这两种方式在设计方法和精确度方面有很大的不同。查找表方式是通过存储器进行设计，设计方法简单，但其精度需要通过提高存储器的深度来实现，且在集成电路中所占面积较大，因此这种方式通常在精度要求不高的近似计算中使用。泰勒级数展开方式采用乘法器和加法器来实现，可以通过增加展开级数来提高计算的精确度。

例 6.2-11 用 Verilog HDL 设计采用查找表方式的 log 函数，输入信号位宽为 4 bit，输出信号位宽为 8 bit。

单目计算函数的特点是，只有一个输入信号和一个输出信号，在结构上与存储器工作原理相似。实现函数操作，是将函数中的计算结果存入存储器中，将输入信号(操作数)作为地址访问存储器，那么存储器输出的结果就是函数的运算结果。由于从存储器中读取数据要比复杂计算的速度快得多，所以通常采用查找表结构，这可以在很大程度上提高运算速度。查找表的方法一般适用于位数比较低的情况，如果位数较高则会占用大量的内存。表 6.2-2 是输入 4 bit、输出 8 bit 信号的 log 函数计算表。

表 6.2-2 log 函数计算表

输入数据	运算结果	输入数据	运算结果
1000	00000000	1100	00011001
1001	00000111	1101	00100000
1010	00001110	1110	00100100
1011	00010101	1111	00101000

其中，输入数据为 1 位整数位与 3 位小数位，精确到 2^{-3}；输出结果为 2 位整数位和 6 位小数位，精确到 2^{-6}。其 Verilog HDL 程序代码如下：

```
module log_lookup(x, clk, out);
    input   [3:0] x;
    input   clk;
    output [7:0] out;
    reg     [7:0] out;
    always@(posedge clk)
      begin
        case(x)
```

```
            4'b1000:out <= 8'b00000000;
            4'b1001:out <= 8'b00000111;
            4'b1010:out <= 8'b00001110;
            4'b1011:out <= 8'b00010101;
            4'b1100:out <= 8'b00011001;
            4'b1101:out <= 8'b00100000;
            4'b1110:out <= 8'b00100100;
            4'b1111:out <= 8'b00101000;
            default:out <= 8'bz;
        endcase
    end
endmodule
//**************************测试代码*******************************//
module log_lookup_tb;
    reg clk;
    reg [3:0]x;
    wire [7:0] out;
    initial
      begin
        x = 4'b1000;
        clk = 1'b0;
        repeat(7)
        #10 x = x+1;
      end
    always #5 clk = ~clk;
    log_lookup U1(.x(x), .clk(clk), .out(out));
endmodule
```

其测试结果如图 6.2-22 所示。

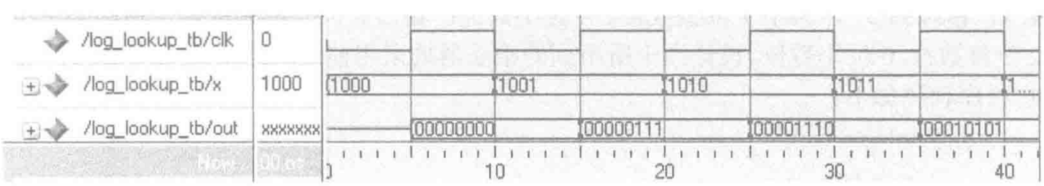


图 6.2-22 查找表方式的 log 函数的测试结果

例 6.2-12 用 Verilog HDL 设计采用泰勒级数展开方式的 log 函数，输入信号位宽为 4 bit，输出信号位宽为 8 bit。

泰勒级数的定义：若函数 f(x)在点的某一邻域内具有直到(n + 1)阶导数，则在该邻域内 f(x)的 n 阶泰勒公式为

$$f(x)=f(x_0)+f'(x_0)(x-x_0)+\frac{1}{2}f''(x_0)(x-x_0)^2 + \cdots+\frac{1}{n!}f^{(n)}(x_0)(x-x_0)^n+\cdots \tag{6.2-3}$$

泰勒级数可以将一些复杂的函数用多项式相加的形式进行近似，从而简化其硬件的实现。$\log_a x$ 在 $x_0=b$ 处的泰勒级数展开为

$$\log_a x=\log_a b+\frac{1}{b}\log_a e(x-b)-\frac{1}{2b^2}\log_a e(x-b)^2+\cdots \tag{6.2-4}$$

误差范围为

$$|R_n|=\left|\frac{1}{(n+1)!}\cdot\frac{(-1)^n}{[b+\theta(x-b)]^{n+1}}\log_a e(x-b)^{n+1}\right| < \frac{1}{(n+1)!}\cdot\frac{1}{b^{n+1}}\cdot\log_a e\,|(x-b)^{n+1}| \tag{6.2-5}$$

在 $x_0=1$ 处展开为

$$\lg x\approx 0.43(x-1)-0.22(x-1)^2 \tag{6.2-6}$$

误差范围为

$$|R_n|<\frac{1}{3!}\cdot\lg e\,|x-1|^3<0.072\ 382 \tag{6.2-7}$$

其电路结构如图 6.2-23 所示。

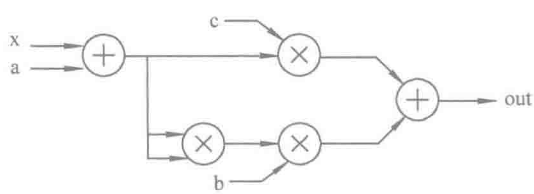


图 6.2-23 泰勒级数展开方式的电路结构

上述的 log 函数在 x = 1 处展开，并且要求 x 的取值范围为 1 < x < 2，输入 4 位二进制数据 x，精确到 2^{-3}，其中 1 位整数位、4 位小数位；输出 8 位二进制数据，精确到 2^{-6}，其中 2 位整数位、6 位小数位。设计当中所用到的乘法器均采用前文所给出的乘法器。其 Verilog HDL 程序代码如下：

```
module log(x, out);
    input[3:0]    x;
    output[7:0]   out;
    wire [3:0]    out1;
    wire [7:0]    out2, out3, out5, out;
    wire [3:0]    out4;
    assign out4 = {out3[7:4]};
    assign out1 = x-4'b1000;                    //x-1
```

```
    wallace U1(.x(out1), .y(4'b0111), .out(out2));              //0.43*(x-1)
    wallace U2(.x(out1), .y(out1), .out(out3));                 //x-12
    wallace U3(.x(out4), .y(4'b0011), .out(out5));          //0.22*(x-1)2
    assign out = out2-out5;                                     //0.43*(x-1)-0.22*(x-1)2
endmodule
//*******************************测试代码**********************//
module log_tb;
    reg [3:0] x = 4'b1000;
    wire [7:0] out;
    log U1(.x(x), .out(out));
    always
    #10 x = x+1;
    always@(x)
      begin
          if(x == 4'b0000)
          $stop;
    end
endmodule
```

其测试结果如图 6.2-24 所示。

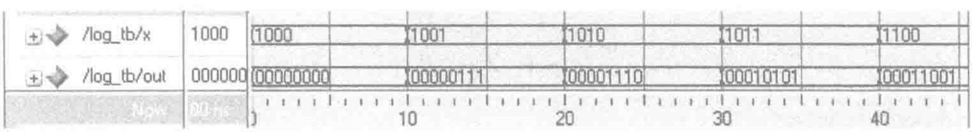


图 6.2-24 泰勒级数展开方式的 log 函数的测试结果

6.2.9 CORDIC 算法的 Verilog HDL 实现

坐标旋转数字计算机 CORDIC(Coordinate Rotation Digital Computer)算法，通过移位和加减运算，能递归计算常用函数值，如 sin、cos、sinh、cosh 等函数。它最早用于导航系统，使得矢量的旋转和定向运算不需要做查三角函数表、乘法、开方及反三角函数等复杂运算。J. Walther 在 1971 年用它研究了一种能计算出多种超越函数的统一算法，引入参数 m 将 CORDIC 实现的三种迭代模式(三角运算、双曲运算和线性运算)统一于同一个表达式下，形成了目前所用的 CORDIC 算法的最基本的数学基础。该算法的基本思想是通过一系列固定的、与运算基数相关的角度不断偏摆以逼近所需的旋转角度。可用下列等式进行描述：

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos\theta_n & -\sin\theta_n \\ \sin\theta_n & \cos\theta_n \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{6.2-8}$$

提出 $\cos\theta_n$，从而得到

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \cos\theta_n \begin{bmatrix} 1 & -\tan\theta_n \\ \tan\theta_n & 1 \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{6.2-9}$$

这里取 $\theta_n = \arctan\left(\frac{1}{2^n}\right)$，所有迭代角度的综合 $\theta = \sum_{n=0}^{\infty} s_n \theta_n$ ，这里 $s_n = \{-1:1\}$，这时矩阵中的 $\tan\theta_n = s_n 2^{-n}$ ，所以矩阵就变为

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \cos\theta_n \begin{bmatrix} 1 & -s_n 2^{-n} \\ s_n 2^{-n} & 1 \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{6.2-10}$$

式中的 $\cos\theta_n = \cos(\arctan\left(\frac{1}{2^n}\right))$。随着迭代次数的增加，该式就会收敛为一个常数：

$$k = \frac{1}{p}\prod_{n=0}^{\infty} \cos(\arctan\left(\frac{1}{2^n}\right)) \approx 0.607\,253 \tag{6.2-11}$$

k 作为一个常数增益，可以暂不考虑，这时上式就会变为

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -s_n 2^{-n} \\ s_n 2^{-n} & 1 \end{bmatrix} \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{6.2-12}$$

如果用 Z 来表示相位累加的部分和，则

$$Z_{n+1} = \theta - \sum_{i=0}^{n} \theta_i \tag{6.2-13}$$

若想使 Z 旋转到 0，则 s_n 的符号由 Z_n 来确定：

$$s_n = \begin{cases} 1 & Z_n \geqslant 0 \\ -1 & Z_n < 0 \end{cases} \tag{6.2-14}$$

旋转后的最终结果为

$$[X_j, Y_j, Z_j] = [p(X_i\cos(Z_i) - Y_i\sin(Z_i)), p(Y_i\cos(Z_i) - X_i\sin(z_i)), 0] \tag{6.2-15}$$

对于一组特殊的初始值：

$$\begin{cases} X_i = \frac{1}{p} = k \approx 0.607\,25 \\ Y_i = 0 \\ Z_i = \theta \end{cases} \tag{6.2-16}$$

得到的结果为$[X_i，Y_i，Z_i] = [\cos\theta，\sin\theta，0]$，这种工作模式称为旋转工作模式。通过旋转工作模式就可以求出一个角度的 sin 和 cos 值。

CORDIC 算法的实现方式主要有两种：迭代结构和流水线结构。

1) 迭代结构

简单地将 CORDIC 算法的公式复制到硬件描述上，就可以实现迭代的 CORDIC 算法，这种结构每个周期都将计算一次式(6.2-8)到式(6.2-16)所示的迭代，其结构示意图如图 6.2-25 所示。

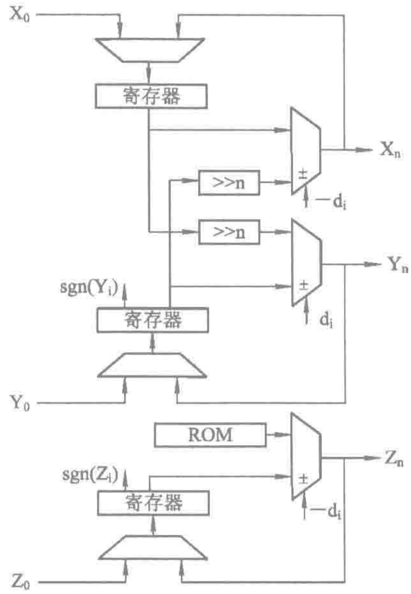


图 6.2-25 CORDIC 算法迭代结构示意图

2) 流水线结构

流水线结构虽然比迭代结构占用的资源多，但是它大大地提高了数据的吞吐率。流水线结构是将迭代结构展开，因此 n 个处理单元中的每一个都可以并行处理一个相同的迭代运算。其结构示意图如图 6.2-26 所示。

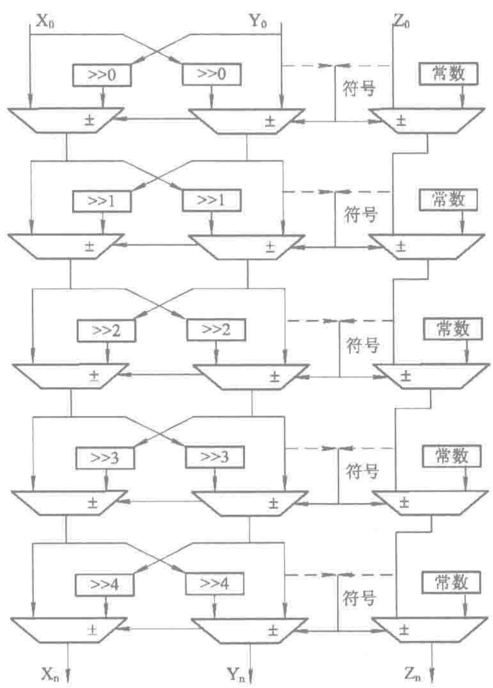


图 6.2-26 CORDIC 算法流水线结构示意图

例 6.2-13　用 Verilog HDL 设计基于 7 级流水线结构求正余弦函数的 CORDIC 算法。

在 CORDIC 算法中有一个初始的 x、y 值，输入变量 z 是角度变量。首先将 x、y 输入到固定移位次数的移位寄存器进行移位，然后将结果输入到加/减法器，根据角度累加器的输出结果来确定加/减法器的加减操作，这样就完成了一次迭代；将此次迭代运算的结果作为输入传送到下一级的迭代运算，将迭代运算依次进行下去，当达到所需要的迭代次数(本例为 7 次)的时候将结果输出，此时就是想要的结果。所以整个 CORDIC 处理器就是一个内部互连的加/减法器阵列。其结构见图 6.2-27。

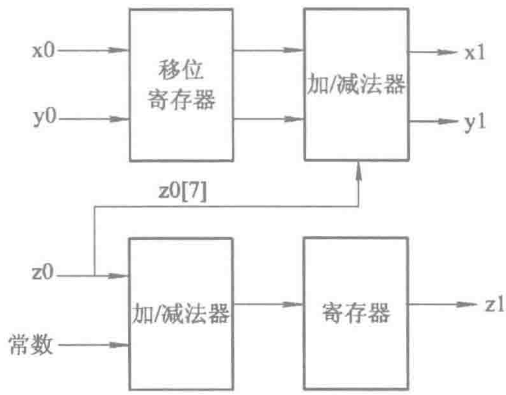


图 6.2-27　运算单元结构图

其 Verilog HDL 程序代码如下：

```
module sincos(clk, rst_n, ena, phase_in, sin_out, cos_out, eps);
    parameter   DATA_WIDTH = 8;
    parameter   PIPELINE = 8;
    input       clk;
    input       rst_n;
    input       ena;
    input       [DATA_WIDTH-1:0] phase_in;
    output      [DATA_WIDTH-1:0] sin_out;
    output      [DATA_WIDTH-1:0] cos_out;
    output      [DATA_WIDTH-1:0] eps;
    reg         [DATA_WIDTH-1:0] sin_out;
    reg         [DATA_WIDTH-1:0] cos_out;
    reg         [DATA_WIDTH-1:0] eps;
    reg         [DATA_WIDTH-1:0] phase_in_reg;
    reg         [DATA_WIDTH-1:0] x0, y0, z0;
    wire        [DATA_WIDTH-1:0] x1, y1, z1;
    wire        [DATA_WIDTH-1:0] x2, y2, z2;
    wire        [DATA_WIDTH-1:0] x3, y3, z3;
    wire        [DATA_WIDTH-1:0] x4, y4, z4;
```

```
wire        [DATA_WIDTH-1:0] x5, y5, z5;
wire        [DATA_WIDTH-1:0] x6, y6, z6;
wire        [DATA_WIDTH-1:0] x7, y7, z7;
reg         [1:0]    quadrant[PIPELINE:0];
integer     i;
 //转换到第一象限
always@(posedge clk or negedge rst_n)
  begin
      if(!rst_n)
        phase_in_reg <= 8'b0000_0000;
      else
      if(ena)
        begin
            case(phase_in[7:6])
                2'b00:phase_in_reg <= phase_in;
                2'b01:phase_in_reg <= phase_in-8'h40; //-pi/2
                2'b10:phase_in_reg <= phase_in-8'h80; //-pi

                2'b11:phase_in_reg <= phase_in-8'hc0; //-3pi/2
                default:;
            endcase
        end
  end
//定义初始常数 x = 0.60725, y = 0
always@(posedge clk or negedge rst_n)
   begin
       if(!rst_n)
         begin
             x0 <= 8'b0000_0000;
             y0 <= 8'b0000_0000;
             z0 <= 8'b0000_0000;
         end
     else
     if(ena)
       begin
           x0 <= 8'h4D;
           y0 <= 8'h00;
           z0 <= phase_in_reg;
       end
```

```
end
Iteration #(8, 0, 8'h20)u1(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x0), .y0(y0), .z0(z0), .x1(x1), .y1(y1), .z1(z1));
Iteration #(8, 1, 8'h12)u2(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x1), .y0(y1), .z0(z1), .x1(x2), .y1(y2), .z1(z2));
Iteration #(8, 2, 8'h09)u3(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x2), .y0(y2), .z0(z2), .x1(x3), .y1(y3), .z1(z3));
Iteration #(8, 3, 8'h04)u4(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x3), .y0(y3), .z0(z3), .x1(x4), .y1(y4), .z1(z4));
Iteration #(8, 4, 8'h02)u5(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x4), .y0(y4), .z0(z4), .x1(x5), .y1(y5), .z1(z5));
Iteration #(8, 5, 8'h01)u6(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x5), .y0(y5), .z0(z5), .x1(x6), .y1(y6), .z1(z6));
Iteration #(8, 6, 8'h00)u7(.clk(clk), .rst_n(rst_n), .ena(ena),
                          .x0(x6), .y0(y6), .z0(z6), .x1(x7), .y1(y7), .z1(z7));
//保留象限信息
always@(posedge clk or negedge rst_n)
   begin
        if(!rst_n)
            for(i = 0; i <= PIPELINE; i = i+1)
            quadrant[i] <= 2'b00;
        else
        if(ena)
           begin
                for(i=0; i <= PIPELINE; i = i+1)
                quadrant[i+1] <= quadrant[i];
                quadrant[0] <= phase_in[7:6];
           end
   end
//输出结果
  always@(posedge clk or negedge rst_n)
  begin
    if(!rst_n)
       begin
            sin_out <= 8'b0000_0000;
            cos_out <= 8'b0000_0000;
            eps <= 8'b0000_0000;
       end
    else
```

```
            if(ena)
            case(quadrant[7])
                2'b00:begin
                    sin_out <= y6;
                    cos_out <= x6;
                    eps <= z6;
                end
                2'b01:begin
                    sin_out <= x6;
                    cos_out <= ~(y6)+1'b1;
                    eps <= z6;
                end
                2'b10:begin
                    sin_out <= ~(y6)+1'b1;
                    cos_out <= ~(x6)+1'b1;
                    eps <= z6;
                end
                2'b11:begin
                    sin_out <= ~(x6)+1'b1;
                    cos_out <= y6;
                    eps <= z6;
                end
            endcase
        end
endmodule
//迭代模块
module Iteration(clk, rst_n, ena, x0, y0, z0, x1, y1, z1);
    parameter   DATA_WIDTH = 8;
    parameter   shift = 0;
    parameter   constant = 8'h20;
    input       clk, rst_n, ena;
    input [DATA_WIDTH-1:0]      x0, y0, z0;
    output[DATA_WIDTH-1:0]      x1, y1, z1;
    reg    [DATA_WIDTH-1:0]     x1, y1, z1;
    always@(posedge clk or negedge rst_n)
      begin
          if(!rst_n)
            begin
               x1 <= 8'b0000_0000;
```

```
                y1 <= 8'b0000_0000;
                z1 <= 8'b0000_0000;
            end
        else
        if(ena)
        if(z0[7] == 1'b0)
            begin
                x1 <= x0-{{shift{y0[DATA_WIDTH-1]}}, y0[DATA_WIDTH-1:shift]};
                y1 <= y0+{{shift{x0[DATA_WIDTH-1]}}, x0[DATA_WIDTH-1:shift]};
                z1 <= z0-constant;
            end
        else
            begin
                x1 <= x0+{{shift{y0[DATA_WIDTH-1]}}, y0[DATA_WIDTH-1:
                        shift]};
                y1 <= y0-{{shift{x0[DATA_WIDTH-1]}}, x0[DATA_WIDTH-1:
                        shift]};
                z1 <= z0+constant;
            end
    end
endmodule
//*************************测试代码***************************
module sincos_tb;
    reg clk, rst_n, ena;
    reg [7:0]phase_in;
    wire [7:0]sin_out, cos_out, eps;
    sincos U1(.clk(clk), .rst_n(rst_n), .ena(ena), .phase_in(phase_in), .sin_out(sin_out),
            .cos_out(cos_out), .eps(eps));
    initial
        begin
            clk = 0;   rst_n = 0;   ena = 1;
            phase_in = 8'b0000_0000;
            #3 rst_n = 1;
        end
    always #5 clk = ~clk;
    always #10
    phase_in = phase_in+1;
endmodule
```

其测试结果如图 6.2-28 所示。

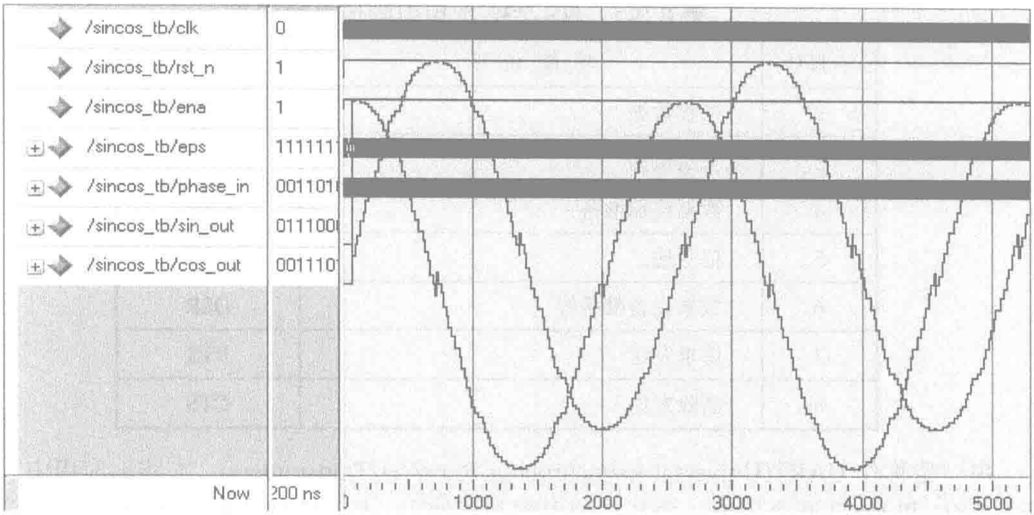


图 6.2-28　正余弦函数的 CORDIC 算法实现的测试结果

6.3 总线控制器设计

总线是主设备与从设备进行数据通信的通道，是数字系统的重要组成部分。本节将对几种常见的总线控制器进行介绍，使大家能够更好地了解数字系统的通信原理。

6.3.1 UART 接口控制器

串行数据接口一般有 RS-232、RS-422 与 RS-485 标准，最初都是由电子工业协会(EIA)制定并发布的。目前 RS-232 是 PC 与通信业中应用最广泛的一种串行数据接口。RS-232 采取不平衡传输方式及单端通信。RS-232 共模抑制能力较差，加上信号线上的分布电容，其传送距离最大约为 15 米，最高速率为 20 kb/s。RS-232 是点对点的通信方式，其驱动负载为 3～7 kΩ。因此 RS-232 适合本地设备之间的通信。

RS-232 最常见的是 9 脚接口，其中常用引脚信号定义如图 6.3-1 所示。

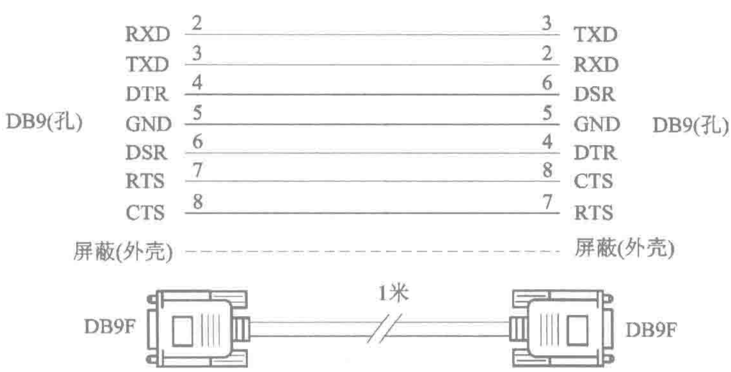


图 6.3-1　RS-232 常用引脚信号定义图

表 6.3-1 给出了 RS-232 常用引脚信号的定义。

表 6.3-1 RS-232 常用引脚信号的定义

引脚号	功能说明	缩写
2	接收数据	RXD
3	发送数据	TXD
4	数据终端准备	DTR
5	信号地	GND
6	数据设备准备好	DSR
7	请求发送	RTS
8	清除发送	CTS

串口也称作 UART(Universal Asynchronous Receiver/Transmitters)。在实际应用中，通常只用 TXD 和 RXD 两个引脚，而其它的引脚都不使用，UART 接口时序如图 6.3-2 所示。

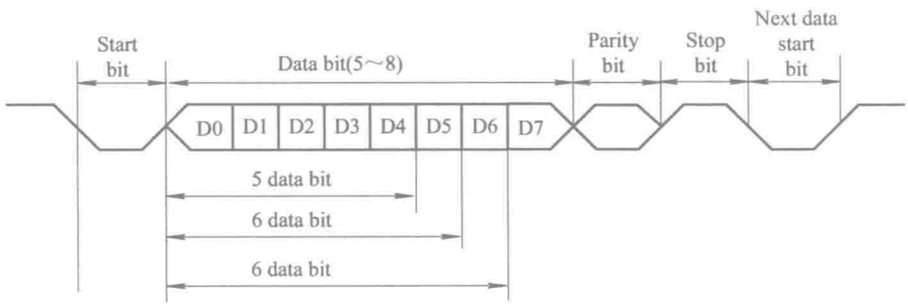


图 6.3-2 UART 接口时序图

在没有数据的情况下接口处于高电平(Mark)，在数据发生前先将电平置低(Space)一个周期，该周期称为起始比特(Start bit)，之后开始发送第 0 个到最后一个比特数据。UART 接口每次可以发送 6 位或 7 位，或 8 位数据。在数据位之后可以有选择地发送一个校验位，可以为奇校验位，也可以是偶校验位。校验位后一个比特是一个或多个停止位，此时数据线回归到高电平，可以进行下一个比特的发送。

一个简单的 UART 结构如图 6.3-3 所示。

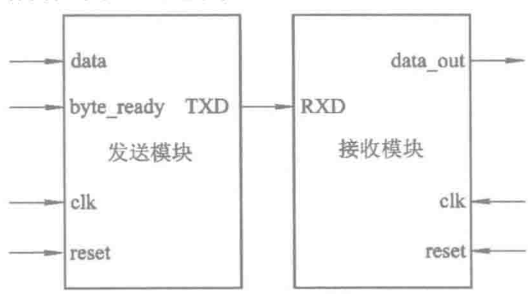


图 6.3-3 UART 结构示意图

发送模块的功能是将数据以串行的形式发送出去，并且将每一组的串行数据加上开始位和停止位。当 byte_ready 信号有效时，数据被载入移位寄存器并添加开始位(低电平)和停

止位(高电平)。当 byte_ready 信号无效时移位寄存器开始移位操作，将数据以串行的形式发送出去。

```
module UART_transmitter(clk, reset, byte_ready, data, TXD);
    input    clk, reset;
    input    byte_ready;                              //数据载入控制
    input    [7:0] data;
    output   TXD;                                     //串行数据
    reg      [9:0] shift_reg;
    assign   TXD = shift_reg[0];
    always@(posedge clk or negedge reset)
       begin
          if(!reset)
          shift_reg <= 10'b1111111111;
          else if(byte_ready)
          shift_reg <= {1'b1, data, 1'b0};            //加起始位与终止位
          else
          shift_reg <= {1'b1, shift_reg[9:1]};        //输出串行数据
       end
endmodule
```

接收模块的功能是接收发送模块输出的串行数据，并以并行的方式将数据送入存储器。当接收模块检测到开始位(低电平)时开始接收数据，并且输入串行数据存入移位寄存器，当接收完成时将数据并行输出。

```
module UART_receiver(clk, reset, RXD, data_out);
    parameter   idle = 2'b00;
    parameter   receiving = 2'b01;
    input    clk, reset;
    input    RXD;                                     //串行数据
    output   [7:0] data_out;
    reg      shift;                                   //移位控制
    reg      inc_count;                               //计数器加控制
    reg      [7:0] data_out;
    reg      [7:0] shift_reg;
    reg      [3:0] count;
    reg      [2:0] state, next_state;
    always@(state or RXD or count)
    begin
       shift = 0;
       inc_count = 0;
```

```
            next_state = state;
            case(state)
                    idle:if(!RXD)next_state = receiving;          //监测起始信号
                    receiving:begin
                        if(count == 8)
                          begin
                            data_out = shift_reg;                 //数据输出
                            next_state = idle;
                            count = 0;                            //计数器清零
                            inc_count = 0;
                          end
                        else
                          begin
                            inc_count = 1;
                            shift = 1;
                          end
                    end
                default:next_state <= idle;
            endcase
    end
    always@(posedge clk or negedge reset)
    begin
    if(!reset)
       begin
          data_out <= 8'b0;
          count <= 0;
          state <= idle;
       end
    else begin
          state <= next_state;
        if(shift)
          shift_reg <= {shift_reg[6:0], RXD};                //接收串行数据
        if(inc_count)
          count <= count+1;
       end
    end
endmodule
//****************************测试代码********************************
module UART_tb;
```

```
    reg clk, reset;
    reg [7:0] data;
    reg byte_ready;
    wire [7:0] data_out;
    wire     serial_data;
    initial
      begin
          clk = 0;
          reset = 0;
          byte_ready = 0;
          data = 8'b10101010;
          #40 byte_ready = 1;
          #50 reset = 1;
          #170 byte_ready = 0;
      end
    always #80 clk = ~clk;
    UART_transmitter U1(.clk(clk), .reset(reset), .byte_ready(byte_ready),
                        .data(data), .TXD(serial_data));
    UART_receiver     U2(.clk(clk), .reset(reset), .RXD(serial_data), .data_out(data_out));
endmodule
```

其测试结果如图 6.3-4 所示。

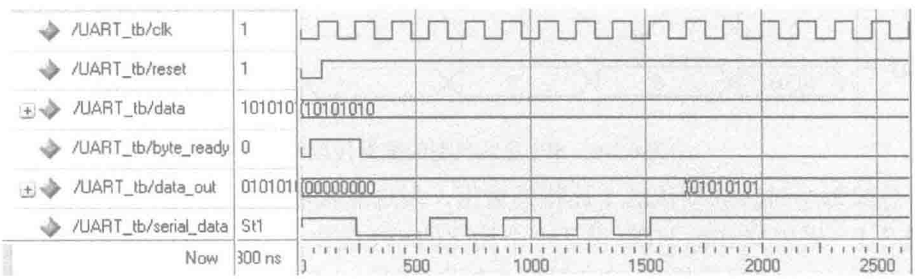


图 6.3-4　UART 接口控制器的测试结果

6.3.2　SPI 接口控制器

串行外设接口(Serial Peripheral Interface，SPI)是一种同步串行外设接口，能够实现微控制器之间或微控制器与各种外设之间以串行方式进行的通信数据交换。SPI 可以共享，便于组成带多个 SPI 接口器件的系统，且传送速率高、可编程、连接线少，具有良好的扩展性，是一种优秀的同步时序电路。

SPI 总线通常有 4 条线：

- 串行时钟线(SCLK)；
- 主机输入/从机输出数据线(MISO)；

- 主机输出/从机输入数据线(MOSI)；
- 低电平有效从机选择线(SS_N)。

SPI 系统可分为主机设备和从机设备两大类，主机提供 SPI 时钟信号和片选信号，从机是接收 SPI 信号的任何集成电路。当 SPI 工作时，在移位寄存器中的数据逐位从输出引脚(MOSI)输出，同时从输入引脚(MISO)逐位接收数据。发送和接收数据操作都受控于 SPI 主设备时钟信号(SCLK)，从而保证了同步。因此只能有一个主设备，但可以有多个从设备，通过片选信号(SS_N)可同时选中一个或多个从设备。其典型结构如图 6.3-5 所示。

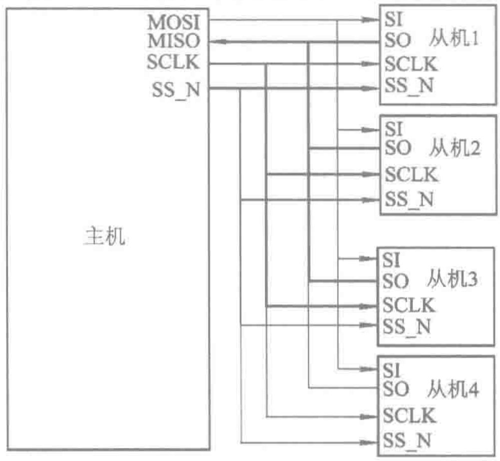


图 6.3-5　SPI 典型结构图

SPI 总线典型的数据传输时序图如图 6.3-6 所示。

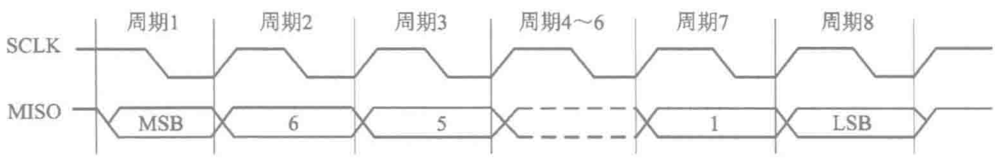


图 6.3-6　SPI 总线典型的数据传输时序图

SPI 总线每个时钟周期传送 1 比特的数据，发送和接收数据的顺序是先高位再低位。

例 6.3-1　采用 Verilog HDL 设计一个简化的 SPI 接收机，用来完成 8 bit 数据的传输。SPI 接收机框图如图 6.3-7 所示。

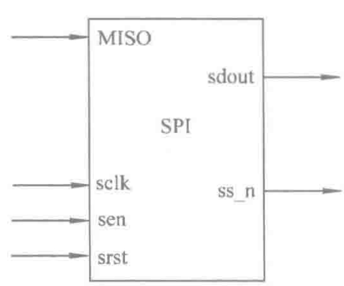


图 6.3-7　SPI 接收机框图

接收机的 Verilog HDL 程序代码如下：

```
module SPI (sdout, MISO, sclk, srst, sen, ss_n);
    output [7:0] sdout;
    output ss_n;
    input MISO, sclk, srst, sen;
    reg [2:0] counter;
    reg [7:0] shift_regist;
    reg ss_n;
    always @(posedge sclk)
```

```
        if (!srst)
          counter <= 3'b000;
        else if (sen)
                  if (counter == 3'b111)
                    begin
                      counter <= 3'b000;
                      ss_n <= 1'b1;
                    end
                  else
                    begin
                       counter <= counter+1;
                       ss_n <= 1'b0;
                    end
              else
                counter <= counter;
      always @(posedge sclk)
        if (sen)
          shift_regist <= {shift_regist[6:0], MISO};
        else
          shift_regist <= shift_regist;
      assign sdout = ss_n?shift_regist:8'bzzzzzzzz;
    endmodule
```

代码中，sclk 为接口时钟；srst 为清零信号，低电平有效；sen 为接口使能信号，高电平有效；ss_n 为片选信号，选择从设备，高电平有效。

当电路上电时，首先将清零信号置为有效，初始化电路。当 sen 使能信号有效后，开始传输数据，由于传输数据为 8 bit，因此 sen 使能信号应至少保持 8 个时钟周期。当 8 bit 数据全部输入后，片选信号 ss_n 有效，选择从设备，将数据整体输出。片选信号 ss_n 由 3 bit 计数器产生，当计数器计数到 111 状态时，ss_n = 1，其它状态下 ss_n = 0。

```
    module SPI_tb;
      reg MISO, sclk, sen, srst;
      wire [7:0] sdout;
      wire ss_n;
         SPI U1 (.sdout(sdout), .MISO(MISO), .sclk(sclk), .srst(srst), .sen(sen), .ss_n(ss_n));
      initial
        begin
          sclk = 0;
          srst = 0;
          MISO = 0;
          sen = 0;
```

```
      #10 srst = 1;
      #10 sen = 1;
      #80 sen = 0;
      #10 sen = 1;
      #80 sen = 0;
    end
  initial
    begin
      #30 MISO = 1;
      #10 MISO = 0;
      #10 MISO = 1;
      #10 MISO = 0;
      #10 MISO = 1;
      #10 MISO = 0;
      #10 MISO = 1;
      #20 MISO = 1;
      #10 MISO = 0;
      #10 MISO = 1;
      #10 MISO = 0;
      #10 MISO = 1;
      #10 MISO = 0;
      #10 MISO = 1;
      #10 MISO = 0;
    end
  always #5 sclk <= ~sclk;
endmodule
```

其仿真测试结果如图 6.3-8 所示。

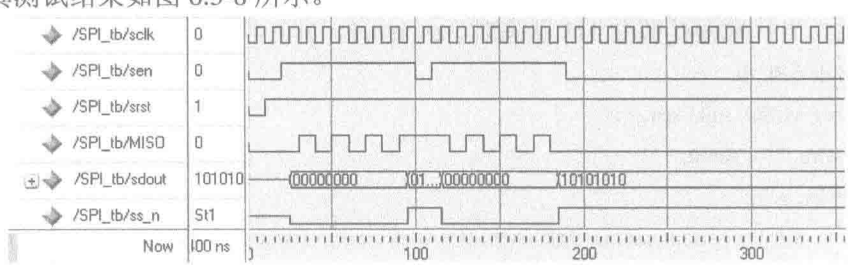


图 6.3-8　SPI 测试结果

本 章 小 结

本章结合数字电路设计的特点，介绍了数字系统的设计方法，对几种较为复杂数字电

路的 Verilog HDL 实现给出了实例，以便读者通过这些电路的设计实例，加深对运用 Verilog HDL 进行数字电路和系统设计的理解。

思考题和习题

1. 集成电路设计中结构性设计方法有几种？简述每种设计方法的特点及其优缺点。

2. 简述层次化描述方法的基本思想，说明层次化描述方法在集成电路设计中的重要作用。

3. 结合 6.2.1 节具体阐述在集成电路设计中如何通过插入寄存器将电路变成流水线型，并简单介绍流水线型电路的特点。

4. 用 Verilog HDL 分别设计 4、5、8 倍分频器。

5. 用 Verilog HDL 设计一个 4 比特的向量叉乘的向量乘法器。

6. 根据 Wallace 树乘法器的原理，用 Verilog HDL 设计一个 5 比特的 Wallace 树乘法器。

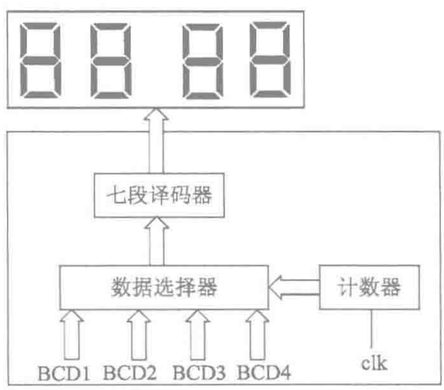
7. 画出可以检测 1010 序列的状态图，采用 Verilog HDL 程序设计语言，用 FSM(有限状态机)进行设计，并写出测试程序。

8. 用查找表方式设计一个 4 比特的乘法器，并说明用查找表方式进行电路设计的优缺点。

9. 用 Verilog HDL 设计一个 4 位 LED 显示器的动态扫描译码电路。要求：

(1) 4 个七段显示器共用一个译码驱动电路；

(2) 显示的数码管清晰明亮，无闪烁现象。



提示：计数器的节拍会直接影响显示的效果，节拍太慢会使显示闪烁；节拍太快会使显示的数码管产生余辉，造成显示不够清晰。因此要合理地选择计数器的工作频率。

10. 设计一个 8 位数字显示的简易频率计。要求：

(1) 能够测试 10 Hz～10 MHz 方波信号；

(2) 电路输入的基准时钟为 1 Hz，测量值以 8421BCD 码形式输出；

(3) 系统有复位键；

(4) 采用分层次分模块的方法，用 Verilog HDL 进行设计。

11. 用 Verilog HDL 设计一个 4 比特的除法器，要求分别采用组合电路与查找表两种不同的方法设计。

12. 用 Verilog HDL 分别设计曼彻斯特编码器和译码器。

13. 根据 SPI 接收机的设计，用 Verilog HDL 设计一个 SPI 发射机。

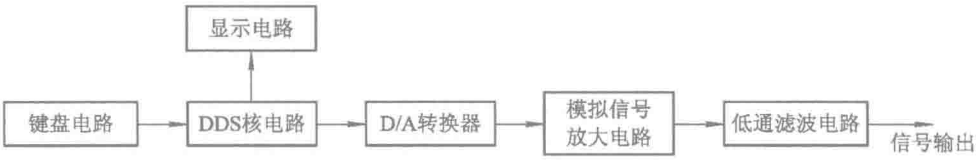
14. 用 Verilog HDL 设计一个 IIC 接口控制电路。

15. 用 Verilog HDL 设计一个 DDS 函数信号发生器。要求：

(1) 利用 DDS 技术产生稳定的正弦波、方波和三角波输出，输出信号的频率为 1 Hz～200 kHz，且频率可调，步进位为 1 Hz、100 Hz、1 kHz 和 10 kHz，峰值为 0～5 V。

(2) 显示电路用来显示输出信号的频率值，也可以显示键盘电路的调整过程。

(3) 用 Verilog HDL 进行建模和模拟仿真。



第 7 章　仿真测试工具和综合工具

7.1　数字集成电路设计流程简介

在 EDA 技术高度发达的今天，没有一个设计工程师队伍能够用人工方法有效、全面、正确地设计和管理含有几百万个门的现代集成电路。利用 EDA 工具，工程师可以从概念、算法、协议等方面开始设计电子系统，通过计算机完成大量的工作，并可以将电子产品从系统规划、电路设计、性能分析到版图、封装的整个过程在计算机上自动完成。使用 EDA 工具有利于缩短设计周期，提高设计正确性，降低设计成本、保证产品性能，尤其是增加了一次投片成功率，因此，EDA 工具在大规模集成电路设计中已经普遍被采用。

利用 EDA 工具进行集成电路设计需要遵循一定的设计流程，这样才能保证设计任务高效率地完成。数字集成电路设计方法的典型流程如图 7.1-1 所示。下面分别介绍各个设计阶段的主要任务。

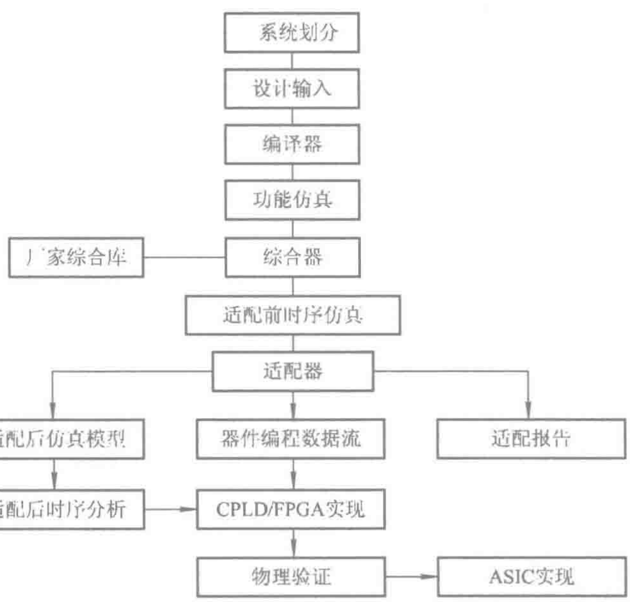


图 7.1-1　集成电路设计的典型流程

7.1.1　设计规范

设计流程从已写出的设计规范开始。设计规范文件是一个包含功能、定时、硅片面积、

功耗、可测性、故障覆盖率以及其它的设计准则的详细说明书。设计规范描述了项目完成的功能，确定设计的总体方案，以平衡各个方面的因素，对整个项目有一个初步的规划。在系统设计阶段，根据对设计面积、功耗、I/O 和 IP 使用等情况的估算，确定所使用的芯片工艺和设计工具。有了设计规范，就可以进行设计划分。

7.1.2 设计划分

设计划分就是把一个复杂设计划分成较小而且较为简单的功能单元。这样一个过程通常被称为自顶向下的设计方法，或者是分层设计法。HDL 可以为需要进行划分、综合和验证的大型复杂系统提供一个通用框架，它支持具有混合抽象级别的自顶向下设计，可以将大型设计中的各部分连接在一起，来进行整个设计的功能和性能验证。

7.1.3 设计输入

设计输入是指将设计划分阶段定义好的模块借助一定的设计输入手段转换为EDA工具能接受的信息格式。目前主要的设计输入手段有高级硬件描述语言 HDL(Verilog HDL/VHDL)和原理图方式。HDL 支持不同层次的描述，不依赖于各个厂家的工艺器件，便于修改。

逻辑输入工具的功能是把逻辑图、状态机、真值表输入到计算机中，并进行语法、综合性检查等。目前的主流工具有 Cadence 公司的 Composer、Synopsys 公司的 Leda 以及 UltraEdit、Vim 等第三方的编辑工具。

Leda 是可编程的语法和设计规范检查工具，它能够对全芯片的 VHDL 和 Verilog HDL 描述或者两者混合描述进行检查，加速 SOC 的设计流程。Leda 预先将 IEEE 可综合规范、可仿真规范、可测性规范和设计复用规范进行了集成，以提高设计者分析代码的能力。

UltraEdit 是一款功能强大的文本编辑器，可以编辑文字、Hex、ASCII 码，内建英文单字检查、C++及 VB 指令突显，可同时编辑多个文件，而且即使开启很大的文件，速度也不会慢。它是一个使用广泛的编辑器，但它并不直接支持 HDL。读者可以在官方网站的链接下载 Verilog HDL/VHDL 的语法高亮文件，并把下载的文件复制到 WordFile.txt 文件中(在 UltraEdit 的安装目录下)，一般加在最后。这样就可以使用 UltraEdit 编辑 HDL 源代码了。

7.1.4 仿真

设计输入完成，经 HDL 编译器检查没有语法错误后，就可以对设计进行验证了。这里的验证是指通过仿真软件验证其功能是否符合制定的设计规范，这一阶段的验证常被称为功能仿真或行为仿真。布局布线后，提取有关的器件延时、连线延时等时序参数(这些信息在反标注文件中)。在此基础上进行的仿真称为后仿真，也称时序仿真，它是接近真实器件运行的仿真。

仿真的结果取决于设计描述是否准确反映了设计的物理实现。仿真器不是一个静态工具，需要 Stimulus(激励)和 Response(输出)。Stimulus 由模拟设计工作环境的 Testbench 产生，Response 为仿真的输出，由设计者确定输出的有效性。

目前，仿真工具比较多，几乎每个公司的 EDA 产品都有仿真工具。Cadence 公司的 NC-Verilog 用于 Verilog 仿真，Mentor 公司推出的是 Verilog 和 VHDL 双仿真器 ModelSim，Synopsys 公司的则是 VSS/VCS 仿真器，这些都是业界广泛使用的仿真工具。

7.1.5 综合

利用综合器对 HDL 代码进行综合优化处理，生成门级描述的网表文件，这是将高层次描述转化为硬件电路的关键步骤。综合优化是针对 ASIC 芯片供应商的某一产品系列进行的，所以综合的过程要在相应的厂家综合库支持下才能完成。

综合实际上是根据设计功能和实现该设计的约束条件(如面积、速度、功耗和成本等)，将设计描述(如 HDL 文件、原理图等)变换成满足要求的电路设计方案，该方案必须同时满足预期的功能和约束条件。对于综合来说，满足要求的方案可能有多个，综合器将产生一个最优的或接近最优的结果。因此，综合的过程也就是设计目标的优化过程，最后获得的结构与综合器的性能有关。这个阶段产生目标 FPGA 的标准单元网表和数据库，供布局布线使用。网表中包含了目标器件中的逻辑元件和互连的信息。

在传统的 IC 设计流程中，前端综合或时序分析时没有精确的线和 CELL 延时信息，这样就容易造成布局前后的时序不收敛。随着工艺的进步，线延时占主导地位，时序收敛问题越来越严重。根本的解决方法是将前后端的设计流程整合起来。物理综合就针对的是这种情况。在 0.18 μm 以下工艺技术的 IC 设计环境下，物理综合将综合、布局、布线集成于一体，让 RTL 设计者可以在最短的时间内得到性能最高的电路。通过集成综合算法、布局算法和布线算法，在 RTL 到 GDS Ⅱ的设计流程中，提供可以确保 IC 设计的性能预估性和时序收敛性。

目前常用的逻辑综合工具有 Synopsys 公司的 Synplify 和 Design Compiler、Physical Compiler，Cadence 公司的 RTL Compiler 等。

7.1.6 适配布线

适配布线就是按照特定的工艺要求和约束条件利用适配器进行布局布线，最后生成版图。对于芯片设计来讲，这个过程通常分 3 步。① 布局规划，主要是标准单元、I/O Pad 和宏单元的布局。I/O Pad 预先给出了位置，而宏单元则根据时序要求进行摆放，标准单元则是给出了一定的区域，由工具自动摆放。布局规划后，芯片的大小，Core 的面积，Row 的形式、电源及地线的 Ring 和 Strip 都确定下来了。② 时钟树生成(Clock Tree Synthesis)。③ 布局布线。

适配完成后，会产生多项设计结果。① 适配报告，包括芯片内部资源利用情况、设计的布尔方程描述情况等。② 适配后的仿真模型。③ 器件编程文件，根据适配后的仿真模型，可以进行适配后的时序仿真。因为此时已经得到器件的实际硬件特性(如时延特性等)，所以此仿真结果能比较精确地预期未来芯片的实际性能。

在 FPGA 设计中各个厂家都提供了相应的布局布线工具，例如 Altera 公司的 Quartus Ⅱ，Xilinx 公司的 ISE 等。在芯片设计领域，有 Cadence 公司提供的 SoC Encounter 和 Synopsys 公司的 Astro 等布局布线工具。

7.1.7 时序分析

时序分析的目的是检查设计中是否有时序上的违规。同步电路的分析采用静态时序分

析实现，异步电路的分析则需要运行特殊仿真激励确认。仿真工具可以用前仿真所用的工具。

静态时序分析(STA)的功能是根据设计规范的要求检查所有可能路径的时序，不需要通过仿真或测试向量就可以有效地覆盖门级网表中的每一条路径，在同步电路设计中快速地找出时序上的异常。

可以识别的时序故障包括：建立/保持和恢复/移除检查(包括反向建立/保持)，最小和最大跳变，时钟脉冲宽度和时钟畸变，门级时钟的瞬时脉冲检测，总线竞争与总线悬浮错误，不受约束的逻辑通道，计算经过导通晶体管、传输门和双向锁存的延时，自动对关键路径、约束性冲突、异步时钟域和某些瓶颈逻辑进行识别与分类。

PrimeTime 是 Synopsys 公司开发的进行静态时序分析(STA)的工具，它可以进行精确的 RC 延时计算、先进的建模和时序验收。对于大型多时钟的设计，比如综合出的逻辑电路、嵌入式存储器和微处理器核的设计，PrimeTime 起到了关键性的作用。

动态时序分析主要是指门级(或对版图参数提取结果)的仿真，它主要应用在异步逻辑、多周期路径、错误路径的验证中。随着设计向 10 nm 以下的工艺发展，只用静态分析工具将无法精确验证串扰等动态效应。通过动态时序分析与静态时序分析的结合，可以验证时序逻辑的建立/保持时间，并利用动态技术来解决串扰效应、动态模拟时钟网络等问题。

7.1.8　物理验证

物理验证通常包括设计规则检测(DRC)、版图与原理图对照(LVS)和信号完整性分析(SI)等。其中 DRC 用来检查版图设计是否满足工艺线能够加工的最小线宽、最小图形间距、金属宽度、栅和有源区交叠的最小长度等要求。如果版图设计违反设计规则，那么极有可能导致芯片在加工的过程中成为废品。LVS 则用来保证版图设计与其电路设计的匹配，保证它们的一致性。如果不一致，就需要修改版图设计。SI 用来分析和调整芯片设计的一致性，避免串扰噪声、串扰延时以及电迁移等问题。

目前主要的物理验证工具有 Mentor 公司的 Calibre、Cadence 公司的 Dracula 和 Diva 以及 Synopsys 公司的 Hercules。此外，各大厂商也推出了针对信号完整性分析的工具。

7.1.9　设计结束

在所有设计约束都已满足，也达到了定时约束条件的情况下，软件就会发出最终设计结束的信号。这时可用于制造集成电路的掩膜集就准备好了。掩膜集的描述是由几何数据(通常为 GDS Ⅱ格式)构成的，这些数据决定了集成电路制造过程中的光掩膜步骤的顺序。

将适配器布局布线后形成的器件编程文件通过下载工具载入到具体的 FPGA 或 CPLD 芯片中，可以方便地实现设计要求。如果是大批量产品的开发则通过更换相应的厂家综合库，便可以转由 ASIC 实现。

7.2　测试和仿真工具

用 HDL 描述完一个硬件系统后要进行仿真验证，而如果想在计算机终端上看到硬件描述语言的输出，则需要通过硬件描述语言的仿真器来完成。常用的 HDL 仿真器有很多种，例如 VCS、NCsim、Verilog HDL-XL、ModelSim、ActiveHDL 等。根据所使用的编程语言

的不同，可以将仿真器分为 Verilog HDL 仿真器和 VHDL 仿真器；根据工作方式的不同，可以分为事件驱动(event-driven)仿真器和时钟驱动(cycle-driven)仿真器等类型。这些工具中，有的侧重于 IC 设计，如 NCsim、VCS 等；有的侧重于 FPGA/CPLD 的设计，如 ModelSim 和 ActiveHDL 等。

ModelSim 在 FPGA/CPLD 设计中应用广泛，这是因为 ModelSim 的出品公司为各种 FPGA/CPLD 厂家提供了 OEM 版本的 ModelSim 工具。ModelSim 可以用于仿真 Verilog HDL，也可以用于仿真 VHDL，同时也支持两种语言的混合仿真。

NCsim(根据使用语言不同，分为 NC-Verilog 和 NC-VHDL)和 VCS 分别由知名的 EDA 工具厂商 Cadence 和 Synopsys 公司提供，在 IC 设计中应用广泛。

本节首先介绍 ModelSim 的使用方法，然后介绍交互方式下 NC-Verilog 的使用。

根据设计阶段的不同，仿真可以分为 RTL 行为级仿真、逻辑综合后门级仿真和时序仿真三大类型。在仿真的后两个阶段，除了 Verilog HDL 源代码外还需要添加两个文件，即工艺厂商提供的库单元文件和延时反标注文件。下面会介绍如何在仿真器中加入这两个文件。

7.2.1 ModelSim 的使用

Mentor 公司的 ModelSim 是一种常用的 HDL 仿真软件，能提供友好的仿真环境，是业界唯一的单内核支持 VHDL 和 Verilog HDL 混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk 技术和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护 IP 核。其个性化的图形界面和用户接口，为用户加快纠错提供了强有力的手段。

ModelSim 的主要特点是：RTL 和门级优化，本地编译结构，编译仿真速度快，跨平台、跨版本仿真；单内核 VHDL 和 Verilog HDL 混合仿真；源代码模版和助手，项目管理；集成了性能分析、波形比较、代码覆盖、数据流 ChaseX、Signal Spy、虚拟对象 Virtual Object、Memory 窗口、Assertion 窗口、源码窗口显示信号值、信号条件断点等众多调试功能；C 和 Tcl/Tk 接口，C 调试；对 SystemC 的直接支持，和 HDL 任意混合；支持 System Verilog 的设计功能；对系统级描述语言的最全面支持，如 SystemVerilog、SystemC、ASIC Sign off。

ModelSim 分为 SE、PE、LE 和 OEM 等几个版本，其中 SE 是最高级的版本，而集成在 Actel、Atmel、Altera、Xilinx 以及 Lattice 等 FPGA 厂商设计工具中的均是其 OEM 版本。SE 版和 OEM 版在功能和性能方面有较大差别，比如对于大家都关心的仿真速度问题，以 Xilinx 公司提供的 OEM 版本 ModelSim XE 为例，对于代码少于 40000 行的设计，ModelSim SE 比 ModelSim XE 要快 10 倍；对于代码超过 40000 行的设计，ModelSim SE 要比 ModelSim XE 快近 40 倍。ModelSim SE 支持 PC、UNIX 和 Linux 平台，提供全面、完善以及高性能的验证功能，全面支持业界广泛应用的标准。

本节介绍的是 ModelSim SE 10.4 版本，它采用用户图形界面操作模式，有许多窗口，如 Main 窗口、Workplace 窗口、Objects 窗口、Wave 窗口、Dataflow 窗口、List 窗口、Source 窗口、Watch 窗口。图 7.2-1 给出了 ModelSim 的常用窗口。

- 主(Main)窗口：ModelSim 唯一的控制窗口，也是控制命令的输入窗口。窗口中显示了 ModelSim 执行仿真的动作以及相应的信息。

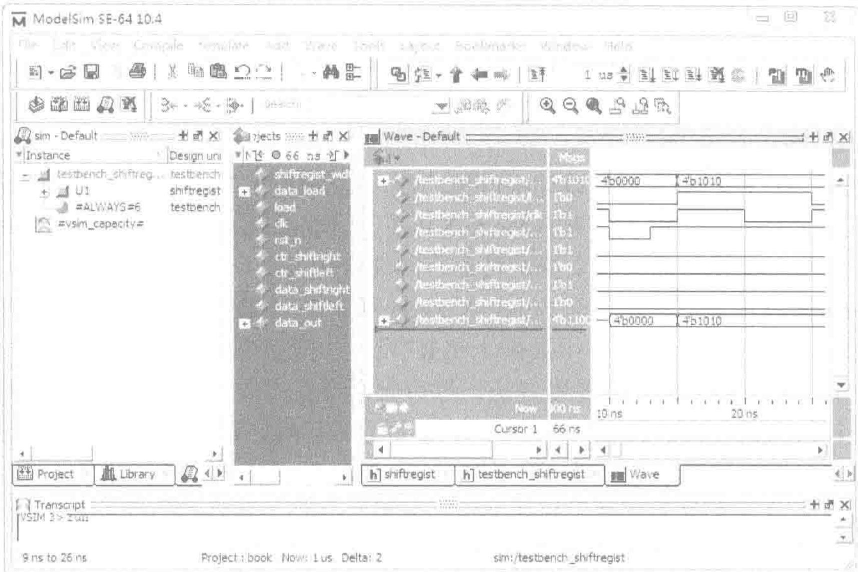


图 7.2-1　ModelSim 常用窗口

- 结构(Workplace)窗口：该窗口按层次关系列出了工程中所有模块之间的关系。在结构窗口中选择固定模块，在信号窗口中会相应地显示这一模块信号的信息。
- 信号(Objects)窗口：显示被选中模块的信号、信号类型以及信号值。
- 波形(Wave)窗口：显示仿真的结果波形。
- 数据流(Dataflow)窗口：用于追踪数据流，并以层次化、图形化的方式显示结果。
- 列表(List)窗口：以表格的形式显示仿真数据。
- 源代码(Source)窗口：显示工程中的相应源代码。
- 观察(Watch)窗口：用于实时监测变量在仿真中的变化情况。

ModelSim 有三种仿真流程：基本仿真流程(Basic Simulation Flow)、工程仿真流程(Project Simulation Flow)和多库仿真流程(Multi-library Simulation)。在较复杂的设计中，推荐使用工程仿真流程，这种流程更容易管理维护设计中遇到的各种类型的文件，事实上工程仿真流程包含了基本仿真流程和多库仿真流程的核心内容，因此本节主要介绍工程仿真流程。

ModelSim 工程仿真流程包括建立库、建立工程、将设计文档(包括源文件和测试文件)加入到工程中并编译、仿真、调试。

1. 建立库并映射

建立并映射库有两种方法。

第一种方法：

在 ModelSim 中选择 File→New→Library 菜单命令，在弹出的对话框中填入库名称，点击 OK 按钮就完成了库的建立和映射。

第二种方法：

在 ModelSim>提示符下运行如下命令：

```
vlib work2
vmap work work2
```

第一条命令实现的是建立新库，第二条命令实现的是映射新库。其中 work2 代表的是新建的工作库。运行完毕后就完成了库的建立和映射操作。

运行完 vlib 命令后会产生 work 库目录，该目录里存放有_info 文件，用于记录各种库中的各种模块。运行完 vmap 命令后会将 ModelSim 安装目录下的 ModelSim.ini 复制到当前的工作目录中并将库和目录对应起来，在[Library]中增加 work = work2 语句。

2. 新建工程项目

选择 Main 窗口的 File→New→Project 菜单命令，新建一个工程。在 Project Name 中输入工程名，在 Project Location 下的对话框中输入保存该工程所有文件的文件夹的路径名。Default Library Name 对话框使用默认设置 work 即可。

3. 输入源代码

选择 Main 窗口的 File→New→Source→Verilog HDL 菜单命令，出现源代码编辑窗口，将源代码输入并保存。源代码文件 shiftregist.v 如下：

```
module shiftregist
  (data_out, clk, rst_n, load, data_load, ctr_shiftright, ctr_shiftleft, data_shiftright, data_shiftleft);
  parameter shiftregist_width = 4;
  output [shiftregist_width-1:0] data_out;
  input [shiftregist_width-1:0] data_load;
  inputload, clk, rst_n, ctr_shiftright, ctr_shiftleft, data_shiftright, data_shiftleft;
  reg [shiftregist_width-1:0] data_out;
  always @(posedge clk or negedge rst_n)
     if (!rst_n)
         data_out <= 0;
     else if (load)data_out <= data_load;
          else if (ctr_shiftright)
                   data_out <= {data_shiftright, data_out[shiftregist_width-1:1]};
               else if (ctr_shiftleft)
                        data_out <= {data_out[shiftregist_width-2:0], data_shiftleft};
                    else data_out <= data_out;
endmodule
```

4. 将文件添加到工程中

刚才输入的文件已经保存在当前 Project 的文件夹中。在 Main 窗口中选择 Project→Add to Project→Existing File...菜单命令，将文件添加到工程中。

5. 编译源代码

在 Workplace 窗口 Project 对话框中选中 shiftregist.v，然后在 Main 窗口中选择 Compile→Compile Selected 菜单命令，对源代码进行编译。编译成功后，Transcript 对话框中将报告“#Compile of shiftregist.v was successful”。如果当前工程中有多个 .v 文件，则可以选择 Compile→Compile All 菜单命令完成对源代码文件的批量编译，也可以一次选择多个文件进行编译。

6. 建立并添加测试文件

用 Verilog HDL 编写测试激励文件，然后进行仿真操作。先输入测试激励文件的源代码，并存盘；然后将该文件添加到当前的工程项目中，再对该文件进行编译。其操作过程与前面介绍的相同。带控制端的移位寄存器的测试激励源代码文件 testbench_shiftregist.v 如下：

```
module testbench_shiftregist;
    parameter shiftregist_width = 4;
    reg [shiftregist_width-1:0] data_load;
    reg load, clk, rst_n, ctr_shiftright, ctr_shiftleft, data_shiftright, data_shiftleft;
    wire [shiftregist_width-1:0] data_out;
    always
      #5 clk = ~clk;
    initial begin data_load = 0; load = 0; rst_n = 1; ctr_shiftright = 0; ctr_shiftleft = 0; clk = 0;
              data_shiftright = 0; data_shiftleft = 0; end
    initial begin #10 rst_n = 0; #3 rst_n = 1; end
    initial begin #15 load = 1; data_load = 4'b1010; #10 load = 0; end
    initial begin #30 ctr_shiftright = 1; #20 data_shiftright = 1; #17 ctr_shiftright = 0;
              #20 ctr_shiftleft = 1; #25 data_shiftleft = 1; #20 data_shiftleft = 0; end
    shiftregist U1(.clk(clk), .rst_n(rst_n), .load(load), .ctr_shiftright(ctr_shiftright),
                   .ctr_shiftleft(ctr_shiftleft), .data_shiftright(data_shiftright),
                   .data_shiftleft(data_shiftleft), .data_load(data_load), .data_out(data_out));
endmodule
```

7. 打开仿真器

在 Main 窗口中选择 Simulate→Start Simulation…菜单命令，得到仿真设置对话框(注意：将当前工作库 work 前面的加号“+”点开，选择 testbench_shiftregist 作为顶层文件进行仿真)。

在 Design 选项卡相应的库名下选择 testbench_shiftregist 模块，再单击 OK 按钮。图 7.2-2 显示的就是打开仿真器后的界面。

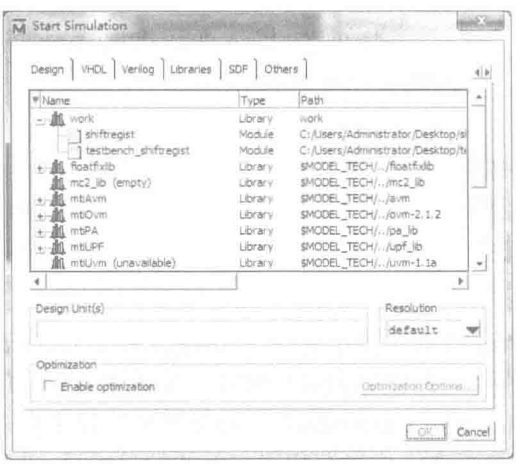


图 7.2-2　打开的仿真器界面

在这里需要注意的是，在 SE 版本中有个使能优化(Enable optimization)选项，建议初学者不要选择优化。因为进行仿真时加载的是一个测试平台，而测试平台的很多语言是会被优化器优化掉的。

8. 打开调试窗口

在 ModelSim 的 Main 窗口的 View 下面有各种全面反映用户设计模块各个方面的特性与内容的窗口，便于用户管理和调试。用户对一个窗口的修改将会自动引起相关窗口的变化，同时用户也可以方便地利用鼠标在窗口之间进行选择和拖放。

要打开窗口，在 Main 窗口的 View 下拉菜单中，单击相应的窗口名即可。已打开的窗口名前有“√”符号提示，再次单击该窗口名将关闭相应窗口，前面的“√”符号也将消失。例如，选择 View→Wave 菜单命令，将打开仿真波形窗口。

9. 添加需要观察的信号

在 Workplace 窗口的 sim 对话框中单击需要观察的模块名，在 Objects 窗口中则会列出该模块的各个端口名及内部信号。可以单击选中其中一个需要观察的信号名，如果按住 Ctrl 键，则可以通过单击选中多个需要观察的信号名，然后选择下拉菜单中的 Add→To Wave →Slected Signals 命令打开 Wave 窗口，则被选中的信号就添加到 Wave 窗口中了。设计者还可以根据调试和测试需要，删除 Wave 窗口中的信号，或向其中添加新的信号。

10. 运行仿真器

在 Main 窗口的下拉菜单 Simulate 选项下有控制仿真器运行的多个命令选项。选择 Simulate→Run 菜单命令，仿真会运行 100 ns(默认的仿真长度)后停止。

在 Main 窗口的 VSIM>提示符下，输入“run 500”，仿真器会再进行另外 500 ns 的仿真，共计仿真了 600 ns。

在主菜单、波形窗口或源代码窗口的工具条上，单击 Run-All 图标，仿真连续运行，直到被中断或在代码中遇到诸如 Verilog HDL 中的$stop 语句等，暂停仿真。若单击 Break 图标，则终止仿真运行。

在 Main 窗口中，选择 Simulate→End Simulation…菜单命令，即可结束仿真。

11. 调试

ModelSim 中的调试手段很多，主要包括：在代码中设置断点、步进调试；观察 Wave 窗口，测量时间；通过 Dataflow 窗口分析物理连接；通过 Memory 窗口观察设计中存储器的数值；统计测试代码覆盖率；波形比较。

Wave 窗口、Dataflow 窗口和 List 窗口是常用的分析手段。

1) 使用 Wave 窗口

观察设计波形是调试设计的一种方法。加载仿真后，就可以使用 Wave 窗口了。通过 View→Wave 菜单命令打开 Wave 窗口，如图 7.2-3 所示。在 Wave 窗口中可以采用多种手段进行调试，如向 Wave 窗口添加项目，对波形显示的图像进行缩放，在 Wave 窗口中使用游标，设置断点，存储波形窗口格式，将当前的仿真结果存储到波形记录格式文件(WLF)中等。

在 Wave 窗口中，游标用于指示仿真的时间位置。当 ModelSim 首次画波形时，自动地

将一个游标放在 0 时刻的位置。使用游标可以测量时间间隔，也可以查找信号跳变的位置。在 Wave 窗口任何位置单击，游标将重新回到鼠标单击的位置。可以对游标进行添加、命名、锁定和删除等操作。

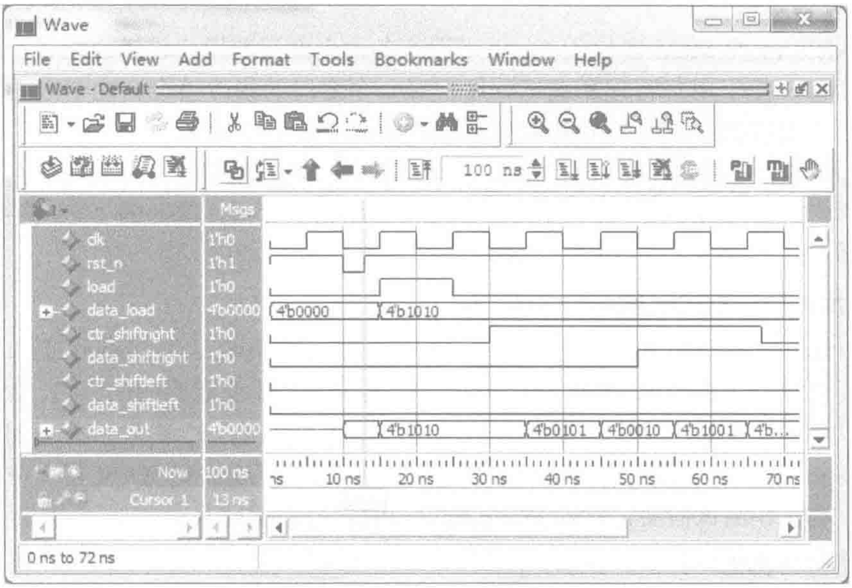


图 7.2-3　Wave 窗口

要养成保存波形文件的良好习惯，以便进行项目的检查和对比。在 ModelSim 中可保存的波形文件及其保存方法如下：

(1) Format 文件。在 Wave 窗口中选择 File→Save Format...菜单命令，在新打开的窗口中填入 DO 文件(仿真脚本文件)的存储路径 E:/shiftregist/wave.do，单击 OK 按钮完成文件的存储。如果需要加载该文件，则在打开的 Wave 窗口中选择 File→Open 菜单命令，然后在弹出的 Open File 窗口中选择 wave.do 文件，即可打开该文件。在 Transcript 对话框中输入该文件的脚本内容并回车，将打开 Wave 窗口，恢复原来的信号、波形和游标状态。

(2) WLF 文件(Datasets)。ModelSim 的仿真结果也可以存储到一个波形格式记录文件中，用于以后浏览和与当前的仿真结果进行比较。通常使用术语“Dataset”表示已创建并可重加载的 WLF 文件。可在主菜单中选择 File→Datasets→Save as 命令，在 Save as 对话框中输入要保存的波形文件名称，点击 OK 按钮就完成了波形文件的保存。选择 File→Datasets→Open 菜单命令，在弹出的 Open Dataset 对话框中，在 Browse 中输入 Datasets 的路径，即可打开已保存的波形文件。

(3) VCD 文件。VCD 文件是 IEEE1364 标准(Verilog HDL 标准)中定义的一种 ASCII 文件。它是一种 EDA 工具普遍支持的通用波形信息记录文件。关于如何使用 Verilog HDL 中的系统函数来生成 VCD 文件，请参考第 5 章内容。

2) 使用 Dataflow 窗口

Dataflow 窗口能够对 VHDL 信号或者 Verilog HDL 的线网型变量进行图示化追踪。双击 Wave 窗口中需要追踪的信号即可打开 Dataflow 窗口，如图 7.2-4 所示。

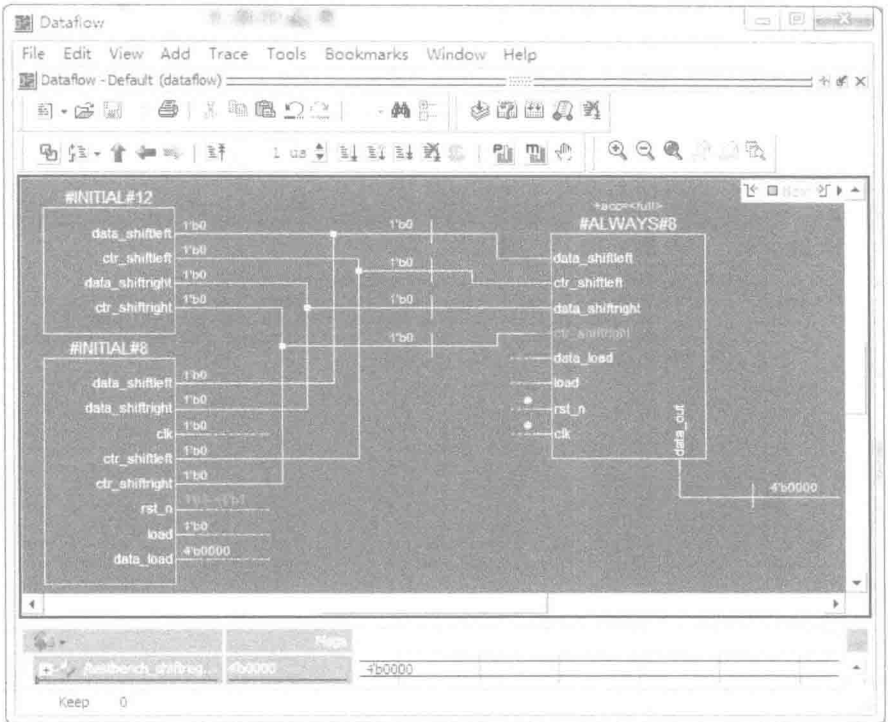


图 7.2-4　Dataflow 窗口

Dataflow 窗口有以下四个功能：

(1) 观察设计的连续性。可以检查设计的物理连接性，可以逐个单元地观测所关注的信号、互连网络或寄存器的输入/输出情况。

(2) 追踪事件。跟踪一个非预期输出的事件。使用嵌入波形观察器，可以由一个信号的跳变回溯追踪，查到事件源头。

(3) 追踪未知态。数据流窗口追踪不定态的功能是工程师比较青睐的。在 Dataflow 窗口中使用 Trace→ChaseX 功能，会不断往驱动级追踪不定态传递的源头。当选择 ChaseX，图形界面不再变化时，就是不定态的源头了。之后就可以根据 Dataflow 窗口的结果，去定位源代码产生不定态的语句，并加以改正。

(4) 显示层次结构。可以使用层次化实例显示设计的连通性。

3) 使用 List 窗口

List 窗口以表格化的方式显示数据，可以方便地通过搜索特殊值或者特定条件的数据，简化分析数据的过程。选择 View→List 菜单命令打开 List 窗口，如图 7.2-5 所示。在 sim 对话框中单击需要观察的模块名，点击鼠标右键，选择下拉菜单命令 Add→To List→All item in region 可以将所有信号添加到 List 窗口，在窗口的左边显示的是仿真的时间点，右边显示的是每个时间点对应的变量值。

当要搜索特定的数值时，可在 List 窗口中选择 Edit→Search...菜单命令，在弹出的 List Signal Search 对话框中选择 Search Type 下的 Search for Signal Value 项，并输入想要搜索的数值。使用这种方式可以很方便地查找仿真中的特殊值，还可以通过这种方式确定特殊值

在什么时间点发生，并以此时间点在 Wave 窗口中定位相应的波形。除此之外还可以设置条件搜索，方法是在 Search Type 下的 Search for Expression 中点击后面的 Builder 建立搜索条件。条件可以是信号、事件、边沿等。

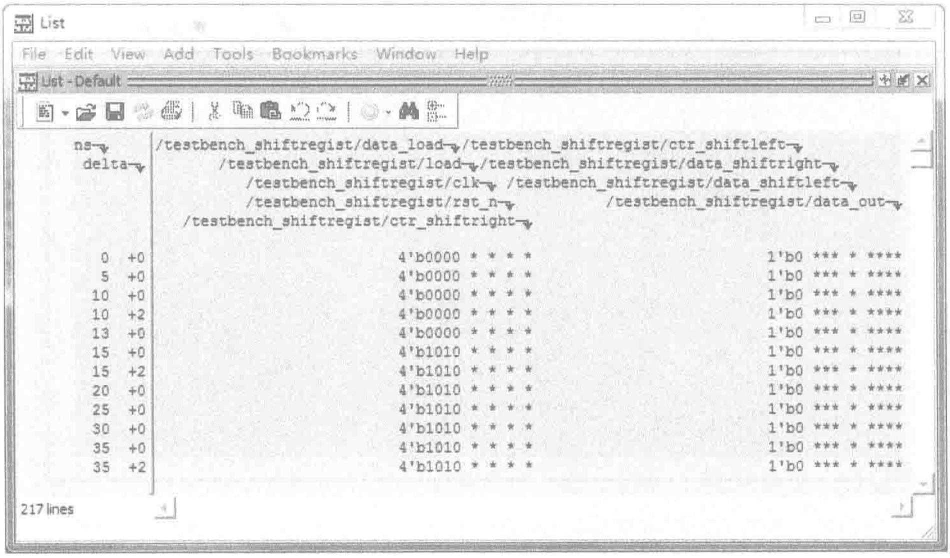


图 7.2-5　List 窗口

与 Wave 窗口一样，List 窗口可以保存数据的列表格式和列表内容。在 List 窗口中选择 File→Save Format 菜单命令，在弹出的 Save Format 对话框中输入所保存列表的名称，再点击“保存”按钮就可以了。数据列表文件也是一个后缀为.do 的可执行脚本文件，可以通过命令 do file_name.do 打开列表文件。在 File→Write List 选项下选择一种格式，可完成对列表内容的保存。列表内容文件是.lst 格式的文件，要查看文件内容可通过记事本打开该类型文件。

12. ModelSim 常用交互命令

ModelSim 的图形化界面提供了多种指令，既可以是单步指令，通过在主窗口的命令窗口中输入命令，也可以构成批处理文件(如 DO 文件)，用来控制编辑、编译和仿真流程。

下面介绍 ModelSim 中用于仿真的一些常用命令，其它指令可参考 ModelSim 说明书或帮助。

1) run 指令

指令格式：

run[<timesteps>] [<time_unit>]

其中，参数 timesteps(时间步长)和 time_unit(时间单位)是可选项，time_unit 可以是 fs(10^{-15}s)、ps(10^{-12}s)、ns(10^{-9}s)、ms(10^{-6}s)、sec(1 s)等几种。

例如，“run”表示运行；“run 1000”表示运行 1000 个默认的时间单元(ps)；“run 3500ns”表示运行 3500ns；“run-continue”表示继续运行；“run-all”表示运行全程。

2) force 指令

指令格式：

force <item_name><value> [<time>]，[<value>][<time>]

其中，参数 item_name 不能默认，它可以是端口信号，也可以是内部信号，而且还支持通配符号，但只能匹配一个；参数 value 也不能默认，其类型必须与 item_name 一致；time 是可选项，支持时间单元。

例如，“force clr 1 100”表示经历 100 个默认时间单元延时后为 clr 赋值 1；“force clr 1，0 1000”表示为 clr 赋值 1 后，经历 1000 个默认时间单元延时后为 clr 赋值 0。

3) force-repeat 指令

指令格式：

force <开始时间><开始电平值>，<结束电平值><忽略时间> -repeat <周期>

指令功能：每隔一定的周期(period)重复一定的 force 命令。该指令常用来产生时钟信号。

例如，“force clk 0 0，1 30-repeat 100”(-repeat 指令可以用-r 替代)表示强制 clk 从 0 时间单元开始，起始电平为 0，结束电平为 1，忽略时间(即 0 电平保持时间)为 30 个默认时间单元，周期为 100 个默认时间单元，占空比为(100 – 30)/100 = 70%。

4) force-cancel 指令

指令格式：

force-cancel < period >

指令功能：执行 period 周期时间后取消 force 命令。

例如，“force clk 0 0，1 30-repeat 60-cancel 1000”表示强制 clk 从 0 时间单元开始，直到 1000 个时间单元结束。

5) view 指令

指令格式：

view 窗口名

指令功能：打开 ModelSim 窗口。

例如，“view source”表示打开源代码窗口，“view wave”表示打开波形窗口，“view dataflow”表示打开数据流窗口。

13. DO 文件

创建 DO 文件就像在文本文件中输入命令一样简单，当然也可以将主窗口的复本保存为 DO 文件。例如，在 ModelSim 中创建一个 DO 文件，在该 DO 文件中添加脚本，实现向 Wave 窗口添加信号，并给这些信号提供激励，而后进行仿真。

选择 File→New→Source→Do 菜单命令，创建一个 DO 文件。在窗口中输入以下命令行：

```
vlib work
vmap work
vlog shiftregist.v testbench_shiftregist.v
vsim -L work -novopt work.testbench_shiftregist.v
add wave -position insertpoint sim:/testbench_shiftregist.v/*
run 2000ns
```

将以上文件保存为 shiftregist.do 文件，每次使用命令 do shiftregist.do 即可自动执行所需的仿真动作。

shiftregist.do 的功能：新建 work 库，将 work 库映射到当前工作目录，编译 shiftregist.v testbench_shiftregist.v 文件(默认编译到 work 库下)，仿真 work 库中名为 testbench_shiftregist 的模块，将 testbench_shiftregist 的所有信号加入到波形图中，运行 2000 ns。

完成移位寄存器仿真批处理文件的编辑后，以 shiftregist.do 为文件名，保存在与计数器设计文件相同的文件夹中，并通过 ModelSim 进行编译。

在 ModelSim 的命令窗口中执行 shiftregist.do 命令，可完成对移位寄存器的仿真。

7.2.2 NC-Verilog 的使用

Cadence NC-Verilog 是业界优秀的 Verilog HDL 仿真器，提供了高性能、高容量的事务/信号视窗和集成的覆盖率分析，并支持 Verilog HDL 2001 特性。NC-Verilog 完全兼容 Incisive 一体化平台(unified platform)，使得用户能够很容易实现纳米(nanometer)工艺集成电路上系统设计的数字化验证。

NC-Verilog 为 Verilog HDL 设计提供了业界优秀的仿真性能，它使用独特的本地 Incisive 一体化仿真器编译架构，从 Verilog HDL 直接生成高效的机器码用于高速执行。

一体化的 NC-Verilog 仿真和调试环境，很容易管理多个设计的运行和分析设计与测试平台。它的事务/波形视窗和原理图追踪器能迅速追踪设计行为到源代码。

NC-Verilog 是全编译仿真器，它直接将 Verilog HDL 代码编译为机器码执行。其过程如下：ncvlog 编译 Verilog HDL 源文件，按照编译指导(compile directive)检查语义及语法，产生中间数据；ncelab 按照设计指示构造设计的数据结构，建立信号连接，产生可执行代码和中间数据；ncsim 启动仿真核，调入设计的数据结构，构造事件序列，调度并执行事件的机器码。

1．工作模式的选择

运行 NC-Verilog 的命令是 nclunch。第一次运行 NC-Verilog 时，需要用户选择工作模式。这里可供选择的主要是 Multiple Step 和 Single Step 模式。请选择 Multiple 模式，此模式对应的仿真流程是 ncvlog、ncelab、ncsim 三步。这两种工作模式在参考手册内有详细说明。图 7.2-6 显示的是 NC-Verilog 的启动界面。

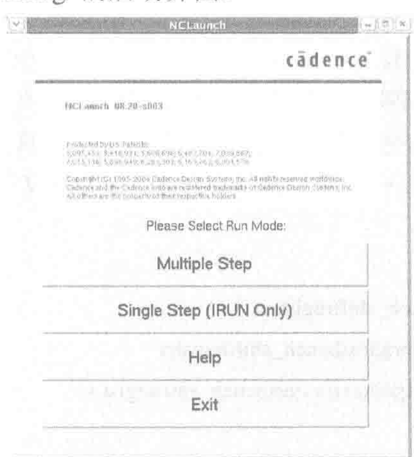


图 7.2-6　NC-Verilog 的启动界面

2. 建立工作环境

作为编译仿真工具，最核心的是源代码部分，由于 NC-Verilog 的源文件编辑界面效果不佳，这里建议用户使用其它支持 VHDL/Verilog HDL 的工具编写代码，在确保没有语法错误后再导入到 NC-Verilog 中进行编译。

选择 File→Set Design Directory 菜单命令进行设置，弹出的对话框如图 7.2-7 所示，会要求填写如下的选项：

- Design Directory：一般就是项目所在的目录，即启动 NCLaunch 时所在的目录。
- Library Mapping File：点击“Create cds.lib File...”按钮，会弹出一个 Create a cds.lib file 对话框。其中的文件名是“cds.lib”，选择“Save”。此时在弹出的对话框中选择“Include Default Libraries”，将会在当前项目目录下建立 INCA_lib 文件夹，用来保存整个设计中全部的库信息。
- Work Library：在建立了 cds.lib 之后将出现 worklib 的默认选项，无需更改。

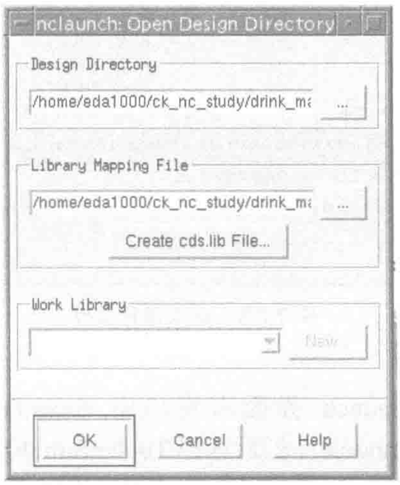


图 7.2-7　建立工作环境

需要说明的是，当完成上述环境设置之后，在工作平台的右上方将出现“Error：Unable to find an hdl.var file to load in”的错误，暂时无需关注此错误。在进行任意文件编译之后软件将自动生成 hdl.var 文件，重新载入工作目录后(File→Set Design Directory)，此错误将消失。

3. 编译

用鼠标左键选择 NCLaunch 界面左窗口中的源文件“shiftregist.v”和“testbench_shiftregist.v”。若有多个设计文件，可以按住 Ctrl 键的同时选择。

第一次编译时选择 Tool→Verilog Compiler 菜单命令，以后可以直接点击工具栏中的“vlog”按钮。

应特别注意的是，在编译成功第一个文件后，应重新载入工作目录(File→Set Design Directory)以确保错误消失。

4. 载入设计文件

用鼠标左键选择 NCLaunch 界面右窗口中工作目录(worklib)下的顶层实体 worklib/

testbench_shiftregist/module(必须提醒的是：如果右边的 worklib 前面没有出现“+”，则请首先检查右边工作平台上对应的的工作路径，确定工作路径已经转换到了和左边一致的目录下)，然后选择 Tool→Elaborator 菜单命令，在弹出的对话框里选中“Access Visibility”的 Read 属性，再确定。载入设计文件后的 NCLaunch 界面如图 7.2-8 所示。

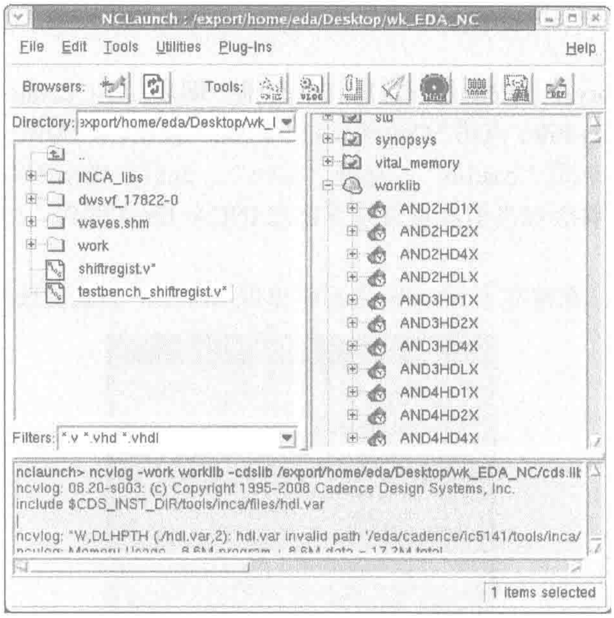


图 7.2-8　载入设计文件

5. 仿真

用鼠标左键选择 NCLaunch 界面右窗口中 Snapshots 下的顶层实体(snapshots /worklib/testbench_shiftregist/module)。然后选择 Tool→Simulator 菜单命令，在弹出的对话框中单击“确定”按钮，弹出仿真器窗口，如图 7.2-9 所示。弹出的默认窗口有两个，点击 Design Browser→SimVision 窗口工具栏中的波形按钮，开启波形仿真窗口。

在 Design Browser→SimVision 窗口内展开左边浏览器中的 Simulator，在右边的列表中选择希望观察的信号。选中后，点击鼠标右键选择 Send to Waveform Windows。

根据实际波形的需要，在 Waveform→SimVision 窗口内把时间显示单位换成 μs、ns 或 ps，然后开始仿真：

(1) 选择 Simulation→Run 菜单命令或者直接点击工具栏中“开始”按钮，开始波形仿真。如果加入了新的信号，需要重新仿真，点击复位，再次仿真。

(2) 在 Console 窗口中直接输入命令，设定仿真时间，例如输入 run 2000 ns。

仿真验证在整个项目设计过程中有着重要的意义。科学合理的仿真方法和仿真技巧可以达到事半功倍的效果；反之，如果只是一味地进行理论分析而不会利用多种工具的优点，则可能会使实际项目寸步难行。

设计者在设计过程中应时刻仿真验证自己的设计。一个系统由很多模块构成，建议每个模块完成后，都要进行完整的仿真测试，不要等到整个系统完成了再整体仿真。这样可以缩短整个设计的周期，提高设计效率。

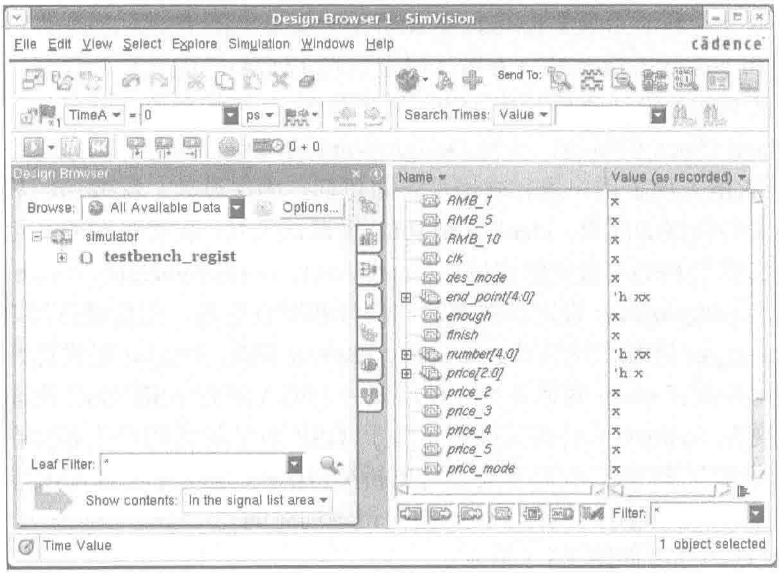


图 7.2-9　仿真器窗口

7.3　综 合 工 具

逻辑综合是前端电路模块设计的重要步骤之一。逻辑综合是在标准单元库和特定的设计约束的基础上，把设计的高层次描述转换成优化了的门级网表的过程。Design Compiler 是 Synopsys 公司用于电路逻辑综合的核心工具，利用它可以方便地将 HDL 描述的电路转换到基于工艺库的门级网表，它是 ASIC 设计领域使用较多的逻辑综合工具之一。FPGA 逻辑综合领域存在着多种逻辑综合工具，其中 Synplify 以其逻辑综合速度快、逻辑综合效果好而备受关注，成为 FPGA 设计逻辑综合的常用工具。

7.3.1　Synplify 的使用

Synplify、Synplify Pro 和 Synplify Premier 是 Synplicity 公司(Synopsys 公司于 2008 年收购了 Synplicity)提供的专门针对 FPGA 和 CPLD 实现的逻辑综合工具。Synplicity 公司的工具涵盖了可编程逻辑器件(FPGAs、PLDs 和 CPLDs)的综合、验证、调试、物理综合及原型验证等方面。

Synplify Pro 是高性能的 FPGA 综合工具，为复杂可编程逻辑设计提供了优秀的 HDL 综合解决方案：其包含的 BEST 算法可对设计进行整体优化；自动对关键路径做 Retiming，可以使性能提高 25%，支持 VHDL 和 Verilog HDL 的混合设计输入，并支持网表*.edn 文件的输入；增强了对 System Verilog 的支持；Pipeline 功能提高了乘法器和 ROM 的性能；有限状态机优化器可以自动找到最优的编码方法；在 Timing 报告和 RTL 视图及 RTL 源代码之间可进行交互索引；自动识别 RAM，避免了繁复的 RAM 例化。

Synplify Premier 是功能强大的 FPGA 综合环境。Synplify Premier 不仅集成了 Synplify Pro 所有的优化选项，包括 BEST 算法、Resource Sharing、Retiming 和 Cross-Probing 等，而

且更是集成了具有专利的 Graph-Based Physical Synthesis 综合技术，并提供 Floor Plan 选项，是业界领先的 FPGA 物理综合解决方案，能把高端 FPGA 性能发挥到最好，从而可以轻松应对复杂的高端 FPGA 设计和单芯片 ASIC 原型验证。这些特有的功能包括：全面兼容 ASIC 代码；支持 Gated Clock 的转换；支持 Design Ware 的转换。同时，因为整合了在线调试工具 Identify，极大地方便了用户进行软硬件协同仿真，确保设计一次成功，从而大大缩短了整个软硬件开发和调试的周期。Identify 是 RTL 级调试工具，能够在 FPGA 运行时对其进行实时调试，加快整个 FPGA 验证的速度。Identify 软件有 Instrumentor 和 Debugger 两部分。在调试前，通过 Instrumentor 设定需要观测的信号和断点信息，然后进行综合，布局布线。最后，通过 Debugger 进行在线调试。Synplify Premier HDL Analyst 提供优秀的代码优化和图形化分析调试界面；Certify 确保客户在使用多片 FPGA 进行 ASIC/SoC 验证时快速而高效地完成工作；现在 Synopsys 公司又推出了基于 DSP 算法的代码产生和综合工具 Synplify DSP，架起了算法验证和 RTL 代码实现之间的桥梁；HAPS 是高性能的 ASIC 原型验证系统，大大减少了流片失败的风险，节省了产品推向市场的时间。

Synplify 软件的界面如图 7.3-1 所示。

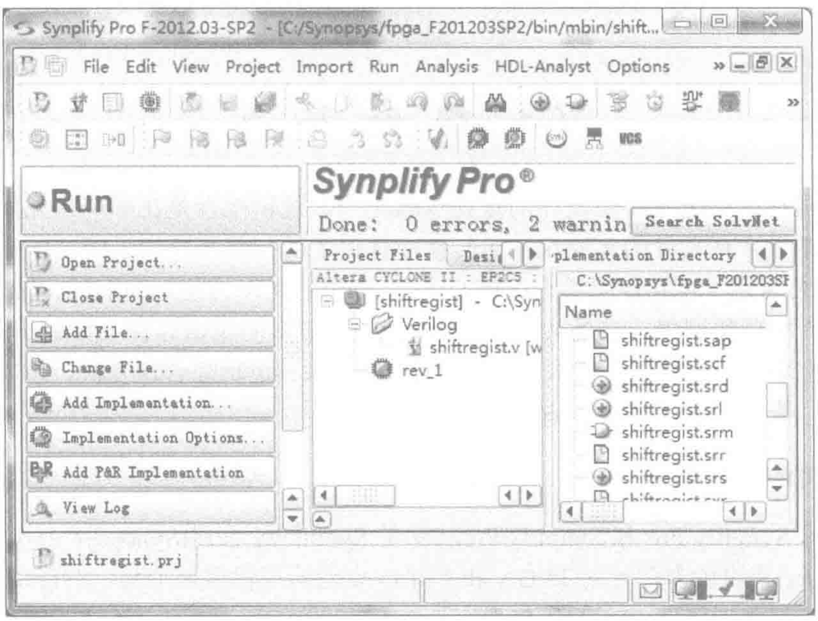


图 7.3-1　Synplify 软件界面

Synplify 软件的具体使用步骤如下。

1. 创建工程

选择 File→New→Project File 菜单命令，然后填入工程名，点击“OK”按钮保存。

创建工程后，其中 rev_1 表示版本一。Synplify 允许对同一个设计根据不同的综合约束条件，创建多个不同的综合版本。

2. 添加文件

接下来的步骤是把设计的源文件添加到工程中。设计文件可以是一个，也可以是多个。添加文件的方法是，选择 Project→Add Source File 菜单命令或者单击界面左边的“Add File...”

按钮，在弹出的对话框中选择要添加的源文件即可。如果事先没有源文件，可以选择新建 VHDL 或者 Verilog HDL 源文件，然后在 HDL 编辑器中编写代码并保存。

3. 保存工程

点击工具栏中的 Save 图标，对工程及源文件进行保存。

4. 语法和综合检测

可以用 Run 菜单中的 Syntax Check 和 Synthesis 对源程序进行检测，检测的结果保存在 Syntax.log 文件中。如果有错误则会用红色标出，双击可以对错误进行定位。

另外，选择 Run→Compiler Only 菜单命令也可以对源文件进行检测。

5. 编译综合前的设置

选择 Project→Implementation Options 菜单命令或者单击界面左侧的“Implementation Options...”按钮，即可出现设置对话框。在设置对话框中设计者可以选择器件、添加一些简单约束等。

在 Device 中选择 Altera CYCLONE II 器件，同时，还可以对与器件映射有关的选项进行设定，包括最大扇出、IO、Pipelining 等。

在 Options 中可以对 Physical Synthesis、FSM Compiler、Resource Sharing、Retiming 等优化选项进行设定。在 Constraints 中可以对时钟频率进行约束，设置时钟频率为 100 MHz。在 Timing Report 中可以设定关键路径的数量。在 Verilog HDL 中的 Top Level Module 中填入 shiftregist。

6. 编译

选择菜单 Run→Compiler Only 菜单命令就可以对设计进行单独编译。在编译后产生的文件中，扩展名为“.srr”的文件是工程报告文件，包括工程检错、编译、综合和时序等所有工程信息；扩展名为“.tlg”的文件是工程组织结构信息文件；扩展名为“.srs”的为 RTL 视图文件，是设计者经常要检查的，双击该文件或者点击工具栏的图标，会出现如图 7.3-2 所示的移位寄存器的 RTL 视图。

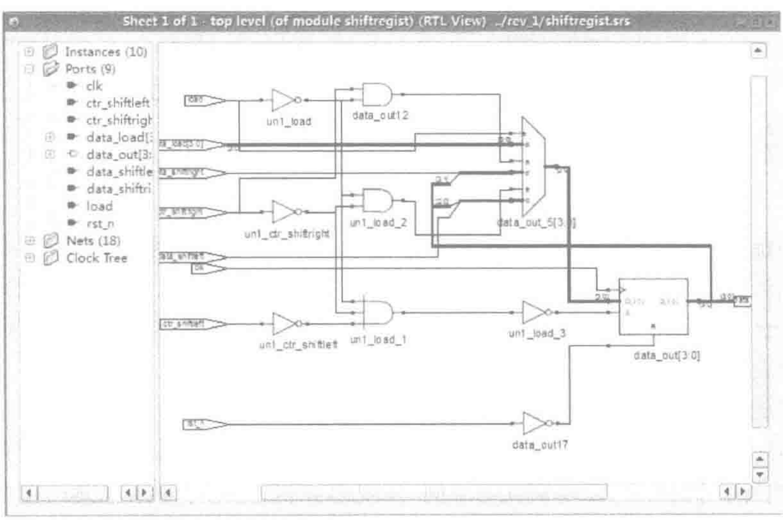


图 7.3-2　移位寄存器的 RTL 视图

RTL 视图由两部分组成，左边区域是模块、结构的分类目录，包括例化(Instances)、端口(Ports)、网线(Nets)和时钟树(Clock Tree)四部分。Instances 目录下是本工程所有调用的模块的实例名和硬件原语，Ports 是模块各个层次 IO 端口的组织关系，Nets 是模块中所有连线名称，Clock Tree 是以树状结构图显示的时钟的依赖关系。RTL 视图具有强大的 CrossProbing 互连切换功能，通过双击 RTL 视图的某一模块，可以连接到产生该模块的 RTL 源代码处，这有利于用户理解代码与硬件的对应关系，方便调试。

7．综合

选择 Run→Synthesize All 菜单命令或者单击面板上的 Run 按钮，就进行了综合。综合后主要产生设计的门级网表，门级网表可以拿到布局布线工具中进行设计的最后实现。综合后还会产生一些其它文件，包括综合报告、Log 文件、脚本文件等。综合后已经根据所选的器件产生了门级电路，设计者可以通过 Technology 视图功能观察门级电路，方法是选择 HDL-Analyst→Technology→Hierarchical View 菜单命令，或者点击工具栏的图标即可。使用这种方法可以查看层次结构显示的与工艺相关的综合结果。移位寄存器的 Technology 视图如图 7.3-3 所示。

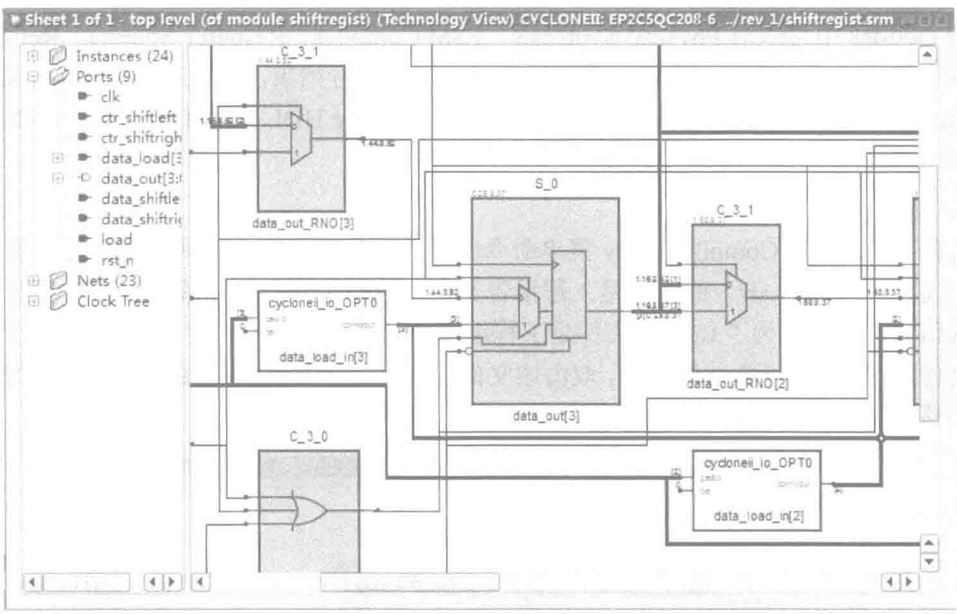


图 7.3-3　移位寄存器的 Technology 视图

在 Technology 视图下可以利用 Push/Pop Hierarchy 层次功能进入硬件的更底层，也可以在此图上显示关键路径；还可以把 Technology 视图展平成门级，方法是选择 HDL-Analyst→Technology→Flattened to Gates View 菜单命令，如图 7.3-4 所示，可查看到门级电路的与工艺相关的综合结果。

需要注意的是，之前介绍的 RTL Viewer 看到的其实是综合工具对 RTL 代码的编译结果，是由基本电路单元连接成的电路，与综合器件无关。由于不同语句、不同方法会导致不同的 RTL 电路，因此 RTL 电路可以用于客观地评价电路的设计效果。在 RTL 视图中看到的不是实际综合出来的结果，只有通过 Hierarchical View 工具才能观测到最后的综合结果。

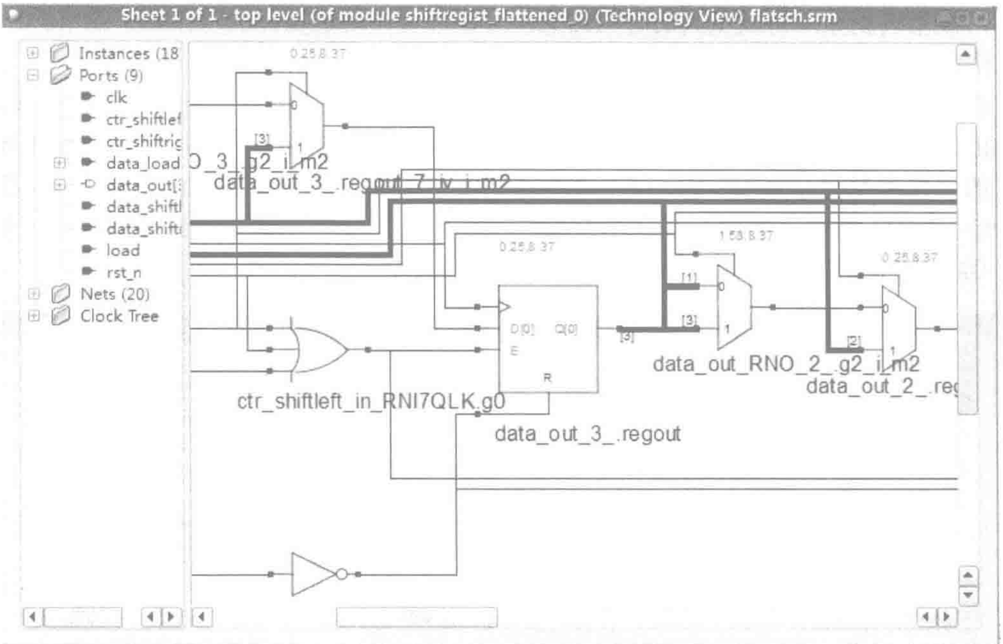


图 7.3-4　移位寄存器的门级电路视图

在综合后生成的“.srr”文件中包含了详细的时序和资源利用的报告，包括时间特性(Timing Report)，即最长延迟时间/最高频率、各个端口的时间信息；面积特性(Area Report)，即器件使用数量(IO 单元、LUT 单元、DSP 块)、门输入数量、节点数量。用户可以通过这个报告，分析设计能够运行的速度和硬件上资源的消耗。但这些都是综合后的估计，更准确的报告需要通过布局布线后才能得出。

8. 分析综合结果

综合通过后，设计者可以点击“View Log”按钮或者双击打开 shiftregist.srr 文件来查看综合报告。综合报告包括了如图 7.3-5 所示的信息。通过查看综合结果，检查系统设计是否满足要求，如系统时钟频率是否达到要求、资源消耗了多少等。同时还可以找出系统设计中存在的问题，如较差路径的起点和较差路径的终点、最差路径等信息。其中，比较重要的是时序方面的报告。

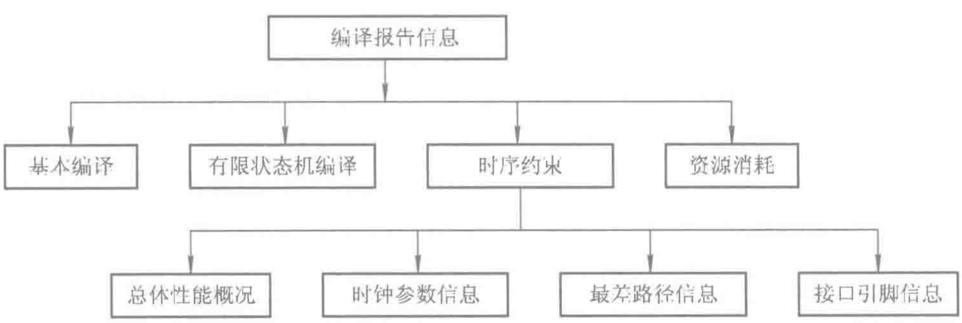
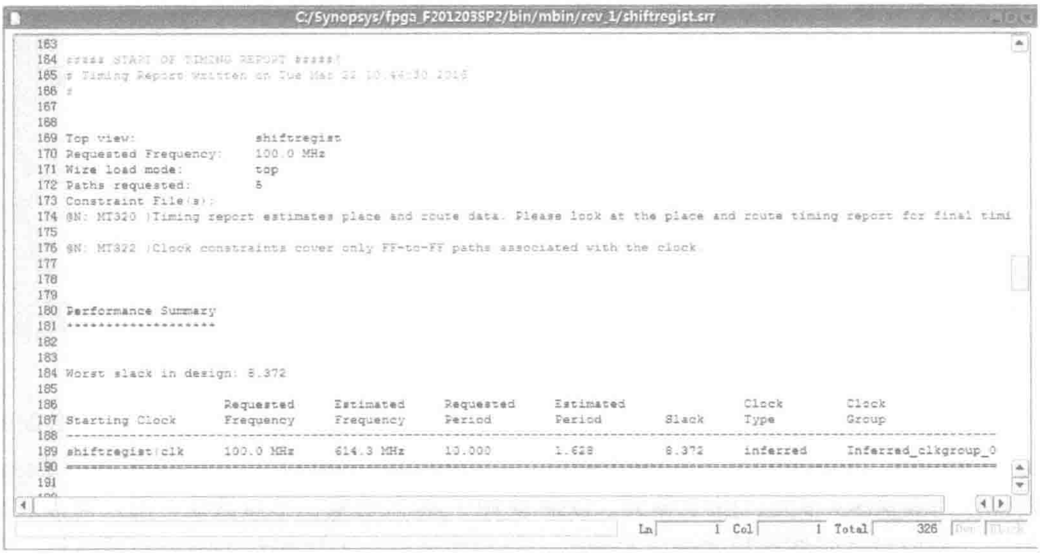


图 7.3-5　综合报告包含的信息

综合后会产生时序报告以及相应的时序估计值。设计的实际时序状况依赖于布局布线

工具，如果调整布局布线工具的时间约束，可以很容易地让设计的工作频率在 10%～20%的范围内变化。

在 Timing Report 中，设计者可以看到用户要求的工作频率(Requested Frequency)和Synplify 综合后系统估计的最高允许的工作频率(Estimated Frequency)。同时也可以看到用户要求的工作周期(Requested Period)、系统估计允许的工作周期(Estimated Period)以及裕量(Slack)。其中，裕量 = 要求周期 – 估计周期。如果裕量大于 0，则满足时序要求；如果裕量小于 0，则不满足时序要求。图 7.3-6 所示的是时序报告。



```
C:/Synopsys/fpga_F201203SP2/bin/mbin/rev_1/shiftregist.srr

##### START OF TIMING REPORT #####[
# Timing Report written on Tue Mar 22 10:44:30 2016
#


Top view:                shiftregist
Requested Frequency:     100.0 MHz
Wire load mode:          top
Paths requested:         5
Constraint File(s):
@N: MT320 |Timing report estimates place and route data. Please look at the place and route timing report for final timi

@N: MT322 |Clock constraints cover only FF-to-FF paths associated with the clock



Performance Summary
*******************

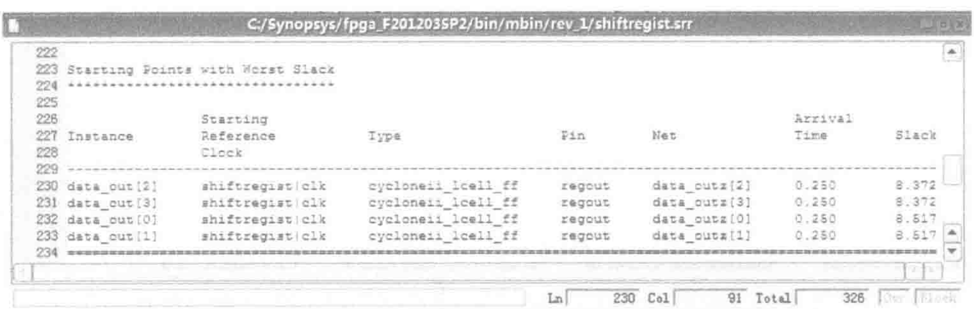

Worst slack in design: 8.372

                      Requested     Estimated     Requested     Estimated                Clock        Clock
Starting Clock        Frequency     Frequency     Period        Period        Slack      Type         Group
-------------------------------------------------------------------------------------------------------------------
shiftregist|clk       100.0 MHz     614.3 MHz     10.000        1.628         8.372      inferred     Inferred_clkgroup_0
===================================================================================================================

Ln 1  Col 1  Total 326
```

图 7.3-6　时序报告

如果裕量小于 0，不满足时序要求时，就要分析长延时路径的起点信息(Starting Points with Worst Slack)，如图 7.3-7 所示。Arrival Time 是时钟从开始端到达该路径终点的延迟时间(也可以说是传播时间)。不符合时序要求的路径裕量会是负值。同样，在终点信息的报告中，相应路径的 Slack 也是最小的。



```
C:/Synopsys/fpga_F201203SP2/bin/mbin/rev_1/shiftregist.srr

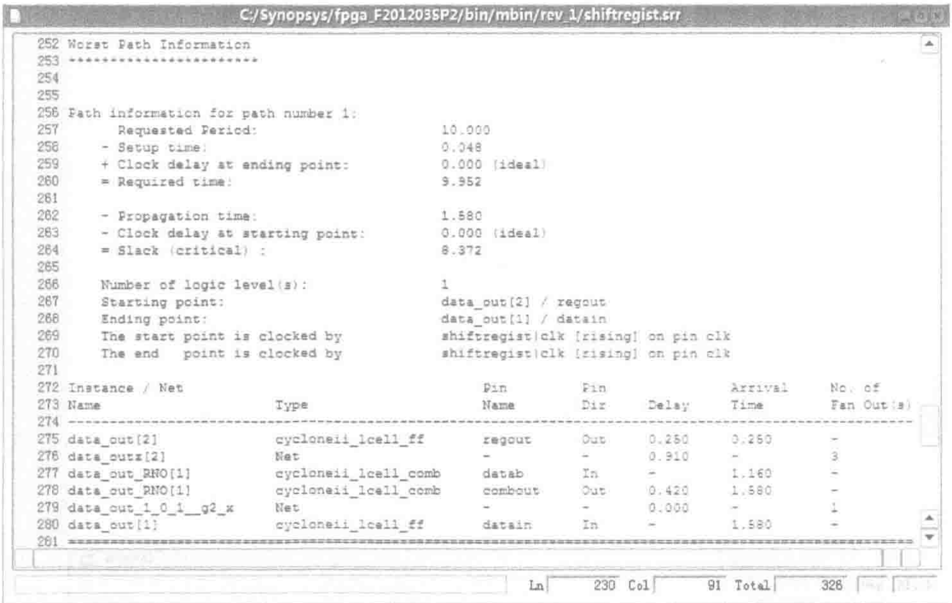
Starting Points with Worst Slack
********************************

                   Starting                                                       Arrival
Instance           Reference           Type                   Pin        Net              Time        Slack
                   Clock
-------------------------------------------------------------------------------------------------------------
data_out[2]        shiftregist|clk     cycloneii_lcell_ff     regout     data_outz[2]     0.250       8.372
data_out[3]        shiftregist|clk     cycloneii_lcell_ff     regout     data_outz[3]     0.250       8.372
data_out[0]        shiftregist|clk     cycloneii_lcell_ff     regout     data_outz[0]     0.250       8.517
data_out[1]        shiftregist|clk     cycloneii_lcell_ff     regout     data_outz[1]     0.250       8.517
=============================================================================================================

Ln 230  Col 91  Total 326
```

图 7.3-7　最长路径起点信息

最差路径信息是对最差路径做的一个总结，它指出了最差路径的时间裕量以及路径的起点和终点。图 7.3-8 显示的是最差路径信息。最差路径通常叫做关键路径(Critical Path)。设计者可以通过分析这些路径，寻找到优化这些路径的方法。



```
C:/Synopsys/fpga_F201203SP2/bin/mbin/rev_1/shiftregist.srr
Worst Path Information
***********************


Path information for path number 1:
      Requested Period:                    10.000
    - Setup time:                          0.048
    + Clock delay at ending point:         0.000 (ideal)
    = Required time:                       9.952

    - Propagation time:                    1.580
    - Clock delay at starting point:       0.000 (ideal)
    = Slack (critical) :                   8.372

    Number of logic level(s):              1
    Starting point:                        data_out[2] / regout
    Ending point:                          data_out[1] / datain
    The start point is clocked by          shiftregist|clk [rising] on pin clk
    The end   point is clocked by          shiftregist|clk [rising] on pin clk

Instance / Net                         Pin        Pin               Arrival     No. of
Name                   Type            Name       Dir     Delay     Time        Fan Out(s)
------------------------------------------------------------------------------------------
data_out[2]            cycloneii_lcell_ff     regout     Out     0.250     0.250       -
data_outz[2]           Net                    -          -       0.910     -           3
data_out_RNO[1]        cycloneii_lcell_comb   datab      In      -         1.160       -
data_out_RNO[1]        cycloneii_lcell_comb   combout    Out     0.420     1.580       -
data_out_1_0_1__g2_x   Net                    -          -       0.000     -           1
data_out[1]            cycloneii_lcell_ff     datain     In      -         1.580       -
==========================================================================================
Ln 230 Col 91 Total 326
```

图 7.3-8　最差路径信息

7.3.2　Design Compiler 的使用

Design Compiler 是 Synopsys 公司综合软件的核心产品，它提供了约束驱动时序最优化选择，并支持众多的设计类型，能把设计者的 HDL 描述综合成与工艺相关的门级设计。它能够从速度、面积和功耗等方面来优化组合电路和时序电路设计，并支持层次化设计。

Design Compiler 按照所有标准 EDA 格式读写文件，包括 Synopsys 内部数据库(.db)和方程式(.eqn)格式。除此之外，Design Compiler 还提供与第三方 EDA 工具的链接，比如布局布线工具，这些链接使得 Design Compiler 和其它工具实现了信息共享。

通过 Design Compiler，设计者可以利用用户指定的门阵列、FPGA 或标准单元库，生成高速、面积优化的 ASIC；能够在不同工艺技术之间转换设计；探索设计的均衡性，包括延时、面积和在不同负载、温度、电压情况的功耗等设计约束条件；优化有限状态机的综合，包括状态的自动分配和状态的优化。

当第三方环境仍支持延时信息和布局布线约束时，可将输入网表和输出网表或电路图整合在一起输入至第三方环境，自动生成和分割层次化电路图。

图 7.3-9 所示是 Design Compiler 的工作界面。

Design Compiler 基本的综合流程包含下列步骤。

1. 编写 HDL 文件

输入 Design Compiler 的设计文件通常都是用诸如 VHDL 和 Verilog HDL 等硬件描述语言编写的。这些设计描述必须小心地编写，以获得可能的最好的综合结果。在编写 HDL 代码时，设计者需要考虑设计数据的管理、设计划分和 HDL 编码风格。划分和编码风格直接影响综合和优化过程。

虽然 Design Compiler 的综合流程中包含该步骤，但实际上它并不是 Design Compiler

的一个步骤。设计者不能用 Design Compiler 工具来编写 HDL 文件。

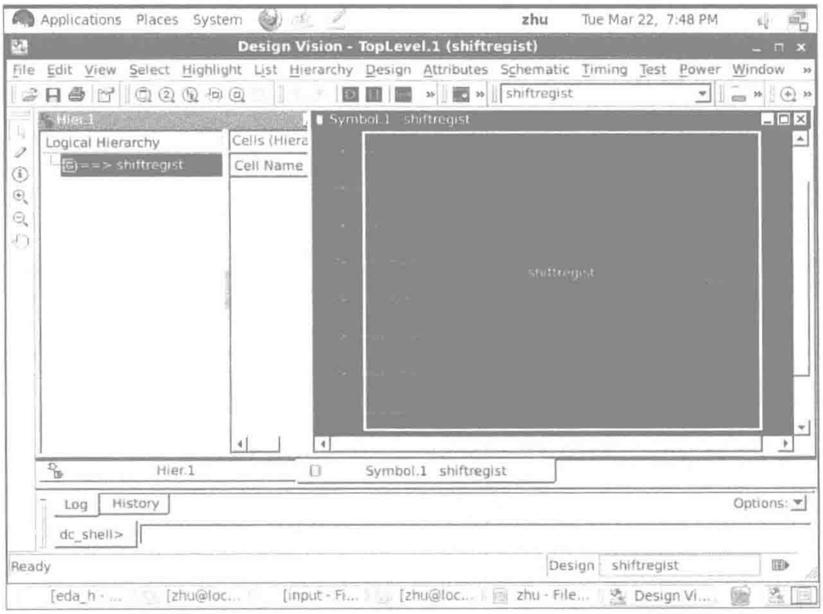


图 7.3-9　Design Compiler 的工作界面

2. 指定库

选择 File→Setup 菜单命令，通过 link_library、target_library、symbol_library 和 synthetic_library 命令为 Design Compiler 指定链接库、对象库、符号库和综合库。

链接库和对象库是工艺库。对象库是指用 RTL 级的 HDL 描述门级时所需要的标准单元综合库，它是由芯片制造商(Foundry)提供的，包含了物理信息的单元模型。链接库可以是同 target_library 一样的单元库，或者是已综合到门级的底层模块设计，其作用如下：在由下向上的综合过程中，上一层的设计调用底层已综合的模块时，将从 link_library 中寻找并链接起来。

符号库定义了设计电路图所调用的符号。如果设计者想应用 Design Analyzer 图形用户界面，就需要这个库。

另外，设计者必须通过 synthetic_library 命令来指定任何一种特殊的有许可的设计工具库(不需要指定标准设计工具库)。

3. 读入设计

Design Compiler 使用 HDL Compiler 将 RTL 级设计和门级网表作为设计输入文件读入。选择 File→Read 菜单命令，在打开的文件对话框中选中要打开的文件，例如选择 shiftregist.v 文件。在 Log 框中出现 successfully 字样表明读入文件成功。有时可能要读入多个文件，每个文件中都有电路 module，则读入后要指定这些 module 中的最顶层 module，例如顶层 module 名是 top，则可以键入命令 current_design top 来指定它。

点击 Symbol 按钮可以查看该电路的 Symbol 图，如图 7.3-10 所示。

如果设计者用 read_file 或 read 命令读入 RTL 设计，等于实现了 analyze 和 elaborate 命令组合的功能。

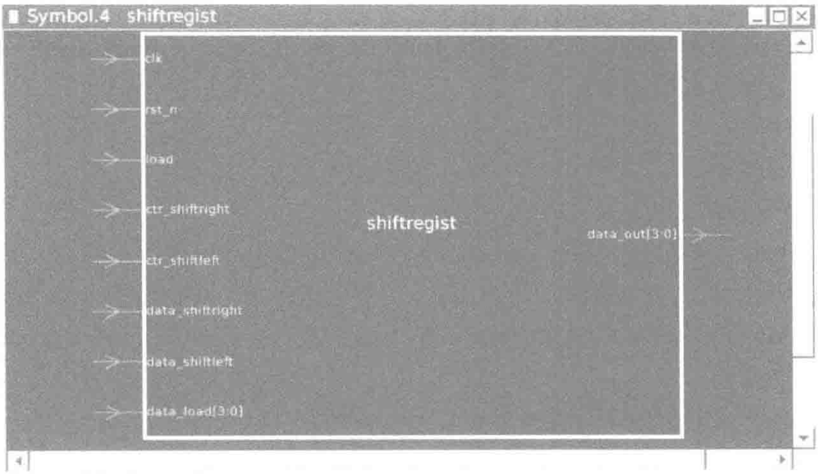


图 7.3-10 电路的 Symbol 图

4. 定义设计环境

Design Compiler 要求设计者模拟出待综合设计的环境。这个模型由外部的操作环境(制造流程、温度和电压)、负载、驱动、扇出、线负载模型等组成，它直接影响到设计综合和优化的结果。

5. 设置设计约束

设计约束定义了时序(时钟、时钟错位、输入延时和输出延时)和面积(最大面积)等设计目标。在最优化过程中，Design Compiler 试图去满足这些目标，但不会去违反任何设计规则。为能够正确地优化设计，必须设置更接近现实情况的约束。

1) 设置时钟约束

在 Symbol 图上选中 clk 端口，选择 Attributes→Specify Clock 菜单命令来设置时钟约束，例如设计者可以进行如下设置：给时钟取名为 clock，周期为 20 ns，上升沿为 0 ns，下降沿为 10 ns。然后点击 OK 按钮，时钟约束设置完成。

以上操作的相应命令是：

```
create_clock -name clock -period 20 -waveform{0 10} [get_ports clk_i]
```

2) 设置复位信号约束

在 Symbol 图中选中 rst_n 端口(在本例中它是复位端口)，选择 Attibutes→Optimization Directives→Input Port 菜单命令。勾选 Don't touch network 选项，然后点击 OK 按钮。

相应的命令是：

```
set_dont_touch_network
```

3) 设置输入信号延时约束

同时选中输入端口，选择 Attributes→Operating Environment→Input Delay 菜单命令。设置 Relative to clock 为 clock(即刚才加约束的时钟信号)，并设置上升延时为 12 ns(根据经验，该值是时钟周期的 60%，本例中设置了时钟周期为 20 ns，20*0.6 = 12 ns)。

相应的命令是：

```
set_input_delay
```

4) 设置输出端口约束

在 Symbol 图上选中输出端口，选择 Attributes→Operating Environment→Output Delay 菜单命令，设置输出延时为 8 ns。

相应的指令是：

set_output_delay

5) 设置面积约束

选择 Attributes→Optimization Constraints→Design Constraints 菜单命令，设置 Max area 的值为 0，表明让 DC 向电路面积为 0 的方向来优化电路，使面积最小。当然，面积为 0 是达不到的。Max fanout 为 4，Max transition 为 0.5(具体含义参见 synthesis.pdf)。

相应的命令是：

set_max_area 0，set_max_fanout 4，set_max_transition 0.5

6. 优化设计

选择 Design→Compile Design 菜单命令，点击 OK 按钮，启动综合和优化进程。

相应的命令是：

compile -map_effort medium

特别地，map_effort 选项可以设置为 low、medium 或 high。

初步编译时，如果设计者想对设计面积和性能有一个快速的了解，则将 map_effort 设置为 low；默认编译时，如果设计者在进行设计开发，则将 map_effort 设置为 medium；当在进行最后设计实现编译时，则将 map_effort 设置为 high。通常设置 map_effort 为 medium。

7. 查看报告

通常 Design Compiler 根据设计综合和优化的结果生成众多的报告，设计者根据诸如面积、约束和时序报告来分析和解决任何设计问题，或者改进综合结果。

选择 Design→Report Area 菜单命令，点击 OK 按钮，查看面积报告，如图 7.3-11 所示。相应的命令为 report_area。报告总面积为 1092.608009，单位是平方微米。

```
****************************************
Report : area
Design : shiftregist
Version: G-2012.06-SP2
Date   : Tue Mar 22 20:02:30 2016
****************************************

Library(s) Used:

    smic25_ss (File: /home/zhu/lib/Synopsys/smic25_ss.db)

Number of ports:                          15
Number of nets:                           39
Number of cells:                          24
Number of combinational cells:            20
Number of sequential cells:                4
Number of macros:                          0
Number of buf/inv:                         6
Number of references:                     11

Combinational area:          619.519997
Buf/Inv area:                 95.743999
Noncombinational area:       473.088013
Net Interconnect area:        undefined  (Wire load has zero net area)

Total cell area:            1092.608009
Total area:                   undefined
```

图 7.3-11　面积报告

选择 Design→Report Constraints，点击 OK 按钮，查看约束报告。相应的命令为 report_constraint -all_violators。从图 7.3-12 可以看出，max_area(最大面积)一项约束不满足。因为我们设置的最大面积约束是 0，而实际综合出的电路面积是 1092.61 平方微米，所以该项 violator 是合理的。

```
****************************************
Report : constraint
        -all_violators
Design : shiftregist
Version: G-2012.06-SP2
Date   : Tue Mar 22 20:02:30 2016
****************************************


   max_area

                          Required        Actual
   Design                   Area           Area            Slack
   -----------------------------------------------------------------
   shiftregist              0.00          1092.61        -1092.61  (VIOLATED)
```

图 7.3-12　约束报告

如果是较为复杂的设计代码，还会有 max_fanout 这一项约束不满足。由于 rst_i 是复位信号，故其扇出非常高，同时由于我们之前对 rst_i 信号设置了 don't_touch，DC 在综合的过程中没有对该信号进行优化，在 DRC 的时候会违反我们设定的 max_fanout(最大扇出)，所以该项 violator 也是合理的。如果还存在其它 violators，说明前面的约束设置不合理或电路设计不合理，需要对其修改。

选择 Design→Report Timing，点击 OK 按钮，查看时序报告。相应的命令为 report_timing。报告的是最大延时路径，这里我们看到图 7.3-13 中所标 slack 值为 6.61，是正值，说明电路满足时序要求。如果该值是负的，就表明电路不满足我们前面所设定的时序约束条件，要更改原设计或调整约束。

```
Point                                        Incr       Path
------------------------------------------------------------
clock clock (rise edge)                      0.00       0.00
clock network delay (ideal)                  0.00       0.00
input external delay                        12.00      12.00 r
ctr_shiftleft (in)                           0.00      12.00 r
U30/Z (OR3HD2X)                              0.36      12.36 r
U29/Z (BUFHD2X)                              0.25      12.61 r
U31/Z (OAI221HD4X)                           0.58      13.20 f
data_out_reg[0]/D (FFDRHDLX)                 0.00      13.20 f
data arrival time                                      13.20

clock clock (rise edge)                     20.00      20.00
clock network delay (ideal)                  0.00      20.00
data_out_reg[0]/CK (FFDRHDLX)                0.00      20.00 r
library setup time                          -0.20      19.80
data required time                                     19.80
------------------------------------------------------------
data required time                                     19.80
data arrival time                                     -13.20
------------------------------------------------------------
slack (MET)                                             6.61
```

图 7.3-13　时序报告

8．保存设计数据

利用 write 命令可保存综合过的设计。Design Compiler 在退出时并不自动保存设计。

在使用 Synplify 或 Design Compiler 对设计进行约束时，如果对电路结构不熟悉，就不要做过多的约束。因为约束越多，使用者就必须对自己的设计越了解，不恰当的约束不仅不能优化综合结果，反而会带来负面影响。

7.4 布局布线工具及后仿真

本节将利用 ModelSim、Synplify Pro 和 Altera 公司的 Quartus Ⅱ这三种工具在 Altera FPGA 上完成对移位寄存器模块的仿真和验证。其工作思路是：① 设计一个带控制端的移位寄存器，利用 ModelSim 做功能仿真；② 利用 Synplify Pro 进行综合，生成 shiftregist.vqm 文件；③ 利用 Quartus Ⅱ导入 shiftregist.vqm 进行自动布局布线，并生成 shiftregist.vo(Verilog HDL Output File)网表文件与 shiftregist.sdo(Stand Delay Output File)延时反标注文件，用作后仿真(Post-Sim)；④ 利用 ModelSim 进行后仿真，看是否满足要求。

在 7.2.1 节已经利用 ModelSim 对 shiftregist 模块做了功能仿真，在 7.3.1 节也已经对 shiftregist 模块利用 Synplify Pro 进行了综合，所以在这一节将直接介绍利用 QuartusⅡ布局布线和后仿真。建议读者自己先新建如下的文件夹：F:\shiftregist\pre-sim、F:\shiftregist\syn、F:\shiftregist\apr、F:\shiftregist\post-sim，以保存相应阶段的文件。仿真所需要的文件有综合生成的网表文件(.vo 文件)、测试激励文件(.v 文件)，以及 Altera 器件库具有时延信息的文件(.sdo 文件)。

7.4.1 工具简介

本小节以 Altera 公司的 Quartus Ⅱ软件为例介绍用于数字集成电路设计的自动布局布线及后仿真流程。Quartus Ⅱ是一个完全集成化的可编程逻辑设计环境，软件界面友好，使用便捷，功能强大，具有开放性、与结构无关、多平台、完全集成化、丰富的设计库、模块化工具等特点，支持原理图、VHDL、Verilog HDL 以及 AHDL(Altera Hardware Description Language)等多种设计输入形式，内嵌综合器以及仿真器，可以完成从设计输入到硬件配置的完整 PLD 设计流程。此外，Quartus Ⅱ通过 DSP Builder 工具与 Matlab/Simulink 相结合，可以方便地实现各种 DSP 应用系统；支持 Altera 的片上可编程系统(SOPC)开发，集系统级设计、嵌入式软件开发、可编程逻辑设计于一体，是一种综合性的开发平台；提供 FPGA 与 Mask-Programmed Devices 开发的统一工作流程。

1．支持的器件

Quartus Ⅱ的不同版本软件可以支持 Altera 公司的 MAX 3000A 系列、MAX 7000 系列、MAX 9000 系列、ACEX 1K 系列、APEX 20K 系列、APEX Ⅱ系列、FLEX 6000 系列、FLEX 10K 系列和 MAX7000/MAX3000 等乘积项器件，并支持 MAX ⅡCPLD 系列、Cyclone、Cyclone Ⅱ/Ⅲ/Ⅳ/Ⅴ、Stratix Ⅱ/Ⅲ/Ⅳ/Ⅴ等，还支持 IP 核，用户可以充分利用成熟的模块，简化设计的复杂性，加快设计速度。

2. 对第三方 EDA 工具的支持

Quartus Ⅱ软件并不直接集成第三方软件，而仅仅提供与这些 EDA 工具的接口。用来实现这种链接的工具便是 NativeLink。NativeLink 能实现第三方工具与 Quartus Ⅱ的无缝交互，双方在后台进行参数行和命令的沟通，使用者完全不必在意 NativeLink 操作的具体细节。它提供给用户的是具有良好互动性的用户界面，设计人员甚至可以在任何一方工具中完成整个操作流程。Quartus Ⅱ集成设计环境对第三方 EDA 工具的良好支持也使用户可以在设计流程的各个阶段使用熟悉的第三方 EDA 工具，如使用 ModelSim 对逻辑进行仿真验证，使用 Synplify Pro 进行逻辑综合。常用第三方 EDA 工具的使用方法可参照 7.2 和 7.3 节。

3. Quartus Ⅱ用户界面

正确安装 Quartus Ⅱ集成设计环境(本小节以 Quartus Ⅱ 13.0 版本为例)后，双击桌面图标，启动软件，进入如图 7.4-1 所示的界面。该用户界面由标题栏、菜单栏、工具栏、资源管理窗口、编辑状态显示窗口、信息显示窗口和工程工作区组成，软件界面友好，符合 Windows 系统应用软件的典型特征。



图 7.4-1　Quartus Ⅱ软件用户界面

- 标题栏：显示当前工程的名称和存储路径。
- 菜单栏：主要由文件(File)、编辑(Edit)、视图(View)、工程(Project)、资源分配(Assignments)、操作(Processing)、工具(Tools)、窗口(Window)和帮助(Help)菜单项组成，各菜单项及其下拉菜单中包含了 Quartus Ⅱ软件的全部核心操作命令。
- 工具栏：部分菜单命令可以通过单击工具栏中的工具按钮来执行。工具栏一旦被打开，用户可以用鼠标将其拖动至任意位置。将鼠标悬停在工具图标上则可显示相应命令。用户也可以选择 Tools→Customize...来定制工具栏按钮。
- 资源管理窗口：显示当前工程中所有相关的资源文件。资源管理窗口左下角有 5 个标签，分别是结构层次(Hierarchy)、文件(Files)、设计单元(Design Units)、知识产权核元件(IP Component)和修正(Revisions)。结构层次窗口在工程编译之前只显示了顶层模块名称，工程编译一次后，此窗口按层次列出工程中的所有模块，并列出每个源文件对应的资源使

用情况。文件窗口列出了工程编译后的所有文件，也可以是图形编辑文件，文件类型有设计器件文件(Design Device Files)、软件文件(Software Files)和其它文件(Others Files)。设计单元窗口列出了工程编译后的所有单元，如 AHDL 单元、Verilog 单元、VHDL 单元等，一个设计文件对应一个设计单元，参数定义文件没有对应的设计单元。

- 工程工作区：器件设置、约束设置、底层编辑器编译报告等均显示在工程工作区中。当 Quartus Ⅱ实现不同功能时，此区域将打开不同的操作窗口，显示不同的内容，进行不同的操作。
- 编辑状态显示窗口：主要显示模块综合、布局布线过程及状态。
- 信息显示窗口：编译或者综合整个过程的详细信息显示窗口，包括编译通过信息和报错信息。如果编译过程因错误而终止，双击信息显示窗口中的错误信息可跳转至相应的代码处，方便查找和排除错误。

7.4.2　布局布线

1．新建工程

首先创建一个工程，用来存放和管理设计过程中编写和生成的文件。启动 Quartus Ⅱ软件，选择 File→New Project Wizard 菜单命令，弹出如图 7.4-2 所示的对话框，单击 Next > 按钮，进入工程路径与名称设置对话框。



图 7.4-2　新工程创建向导

2．选择工程路径与设置工程名称

在图 7.4-3 所示对话框中点击 ... 按钮，选择工程文件夹路径。注意，路径中最好不要出现中文字符，否则可能在后来的布局布线过程中出现意想不到的错误，这里选择已经建立好的路径 F:\shiftregist\apr。在工程名一栏中输入工程名称，这里的工程名取作 shiftregist(工

程的命名最好体现设计内容)。工程名称一旦确定，最后一栏的顶层设计实体名则自动设置为与工程名相同，即 shiftregist，本书建议工程名、顶层设计文件名、顶层设计实体名保持一致。完成上述设置后单击 Next > 按钮，进入下一个设置。



图 7.4-3　工程路径与名称设置对话框

3. 添加设计文件

在对设计进行布局布线操作前，需将之前的逻辑综合结果作为设计文件导入工程，本小节以第三方 EDA 工具 Synplify Pro 为例，将逻辑综合结果 shiftregist.vqm 文件导入。单击图 7.4-4 中的 ... 按钮，找到目标文件，单击 Add 按钮添加。如果现在不做添加文件操作，也可以在后续编辑过程中通过 Assignments→Settings... →Files 菜单命令添加。单击 Next > 按钮，进入器件选择窗口。

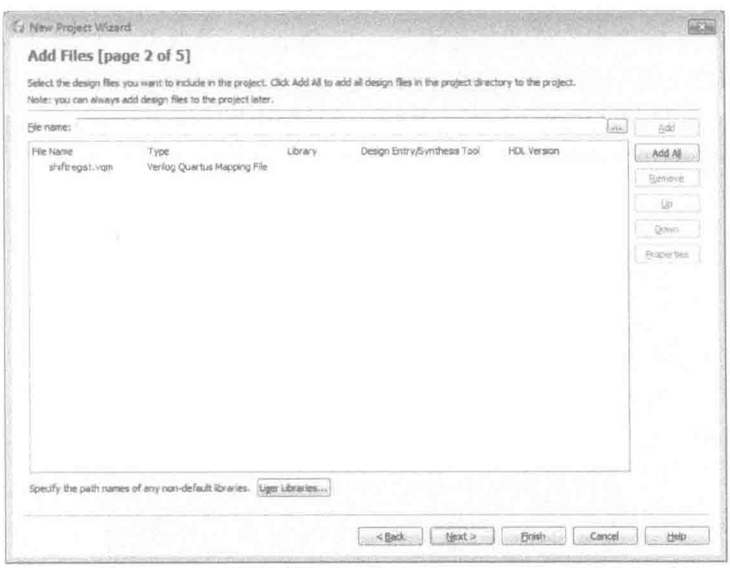


图 7.4-4　设计文件选择窗口

4. 设定 Family 和 Device

Quartus Ⅱ支持众多可编程器件的配置。为了进行物理验证，必须首先指定一个确切的器件，这里选用 Cyclone Ⅱ系列的 EP2C5AF256A7 芯片(具体情况需根据实验箱上 FPGA 芯片的型号来进行选择)。为了迅速找到所需要的型号，如图 7.4-5 所示，首先在 Device family 的 Family 栏下拉列表中选择 Cyclone Ⅱ系列，再通过窗口右上角的封装(Package)、引脚数(Pin count)、速度等级(Speed grade)筛选条件进行选择，也可以在 Name filter 一栏直接输入芯片具体型号进行选择。单击器件目录选中器件，点击 Next > 按钮进入设计工具选择窗口。

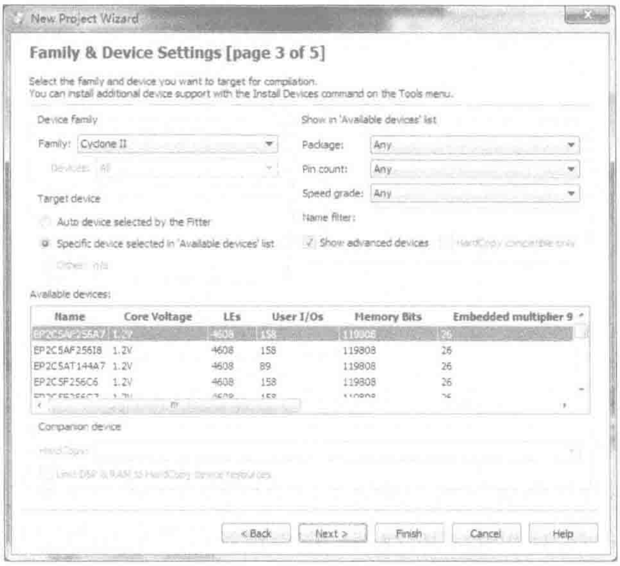


图 7.4-5 器件选择对话框

5. 设定相关 EDA tools

Quartus Ⅱ针对仿真验证、逻辑综合和时序分析等过程提供了第三方软件接口，允许用户使用第三方 EDA 软件来分析和优化设计，与 Quartus Ⅱ互补。如图 7.4-6 所示界面，本小节在输入与综合工具中选择 Synplify Pro，文件格式选择 VQM。在仿真工具的下拉列表中选择 ModelSim，文件格式选择 Verilog HDL。设置完成后单击 Next > 按钮，将对所建立工程的信息和各项设置进行汇总显示。

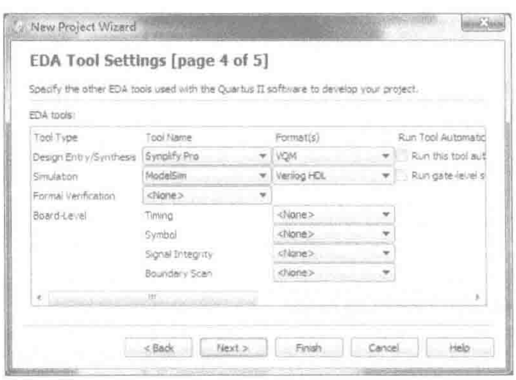


图 7.4-6 综合器与仿真器选择

6．确认信息

在工程信息汇总对话框中可以查看之前设定的所有信息。如果发现问题，可以点击 < Back 按钮返回修改。单击 Finish 按钮则完成工程建立，如图 7.4-7 所示。



图 7.4-7　工程信息显示窗口

7．读取信号端口信息

选择 Processing→Start→Start Analysis&Elaboration 菜单命令，对设计进行分析和检查，同时软件还可读取信号端口信息。

8．引脚锁定

根据硬件接口设计要求对芯片进行引脚锁定。选择 Assignments→Pin Planner 菜单命令，弹出如图 7.4-8 所示的窗口，双击对应引脚后，在 Location 空白框中选择要锁定的引脚，逐次完成所有引脚的分配。

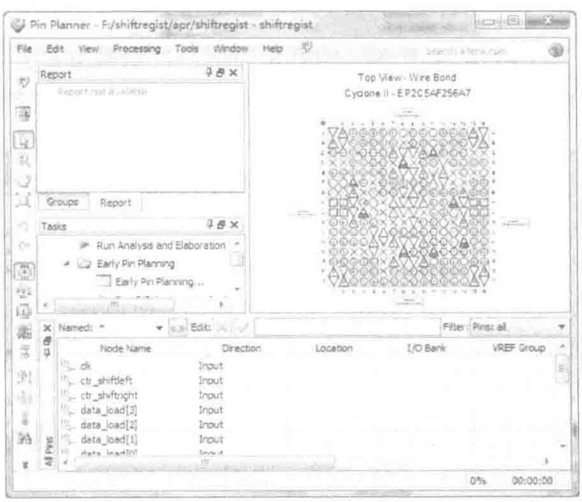


图 7.4-8　引脚分配窗口

选择 Assignments→Device...菜单命令或单击工具栏中的按钮，打开器件设置窗口，选择 Device and Pin Options，弹出图 7.4-9 所示窗口，切换到 Unused Pins 选项卡，对未用引脚进行设置。将 Unsigned Pins 选项卡中的 Reserve all unusned pins 设置为输入三态，然后单击 OK 按钮完成引脚锁定设置。

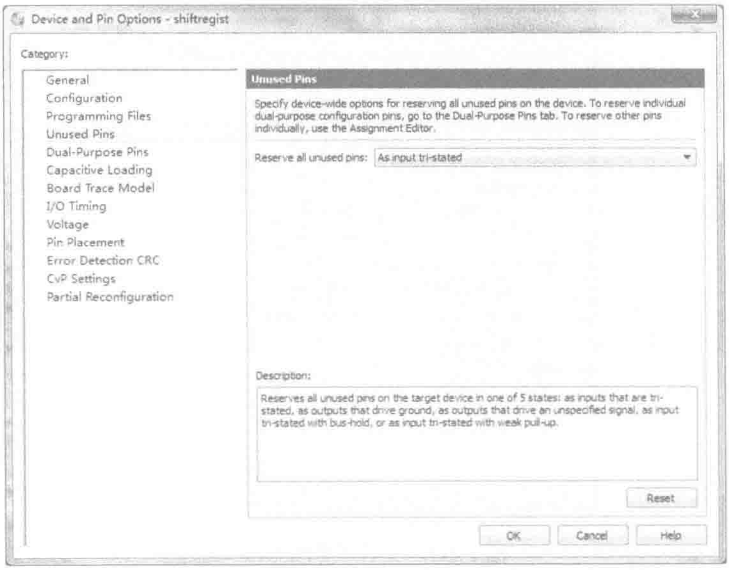


图 7.4-9　未用引脚设置选项卡

9. 全编译

完成上述操作后，选择 Processing→Start Compilation 菜单命令或单击工具栏中的按钮进行编译，即可开始对输入设计文件进行自动布局布线操作。编译成功后，会在 F:\shiftregist\apr 目录下生成 simulation 子目录，设计者可在 F:\shiftregist\apr\simulation\modelsim 目录下找到用于后仿真验证所需要的 shiftregist.vo 和 shiftregist.sdo 文件。

7.4.3　后仿真

本小节以使用第三方 EDA 工具 ModelSim 为例，针对独立使用 ModelSim 的方法，介绍数字集成电路设计的后仿真流程。ModelSim 工具的具体使用方法参照 7.2 节内容。

(1) 新建工程。启动 ModelSim，然后建立一个 Project。建立的方法和前面介绍功能仿真时的方法一样，注意 Project Location 路径选择为 F:\shiftregist\post-sim，Project Name 填为 shiftregist，如图 7.4-10 所示。

(2) 加入文档。首先将用 Quartus Ⅱ生成的网表文件 shiftregist.vo 添加到上述 Project 中。

由于设计采用了 Altera 的 Cell Library 来综合电路，所以综合后的 Netlist 里所包括的那些 Logic Gates 与 Flip-Flop 都是出自于 Altera 的 Cell Library，因此仿真时需添加 Cell Library。仿真库的添加方法见 7.4.4 节。

设计时所选用的 Family 是 Cyclone Ⅱ，所以在 Quartus Ⅱ edasim_lib 里将 Cyclone Ⅱ的 Cell Library(cycloneii_atoms.v 文件)加入。cycloneii_atoms.v 可以在目录 C:\altera\13.0\quartus\eda \sim_lib 下找到。

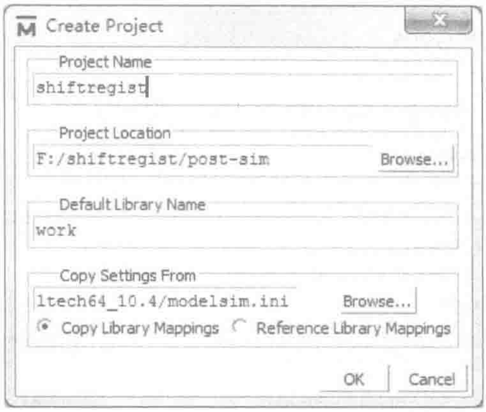


图 7.4-10　新建工程

最后添加测试平台 testbench_shiftregist.v，并在测试平台里加上`timescale 1ns/1ns。

(3) 编译。选择 Compile→Compile All，即可编译工程中的所有文件。

(4) 仿真设置。在 ModelSim 主菜单中，选择 Simulate→Start Simulation…，并在弹出的对话框中做如下设置：首先，在 Design 选项卡下点开 work 前面的加号“+”，选择测试文件 testbench_shiftregist，不勾选 Enable optimization，如图 7.4-11 所示。其次，打开 Libraries 选项卡，在 Search Libraries 项目下点击 Add…按钮，在弹出的 Select Library 对话框中点击 Browse…按钮，找到 work 文件夹的路径并加入，如图 7.4-12 所示。这里因为刚才编译时已经将 cycloneii_atoms.v 库的信息加进了 work，所以添加 work 库的同时也就添加了 Altera 的 Cyclone Ⅱ器件库。最后，在 SDF 选项卡下添加延时反标注文件，点击 Add…按钮，在弹出的 Add SDF Entry 对话框中，点击 Browse…按钮，找到 .sdo 文件的路径并加入。在作用区域（Apply to Region）下所填的是测试模块名和测试模块中例化文件名，格式是“/测试文件顶层模块名/测试文件中例化文件名”，本例中填写为“/testbench_shiftregist/U1”，如图 7.4-13 所示。

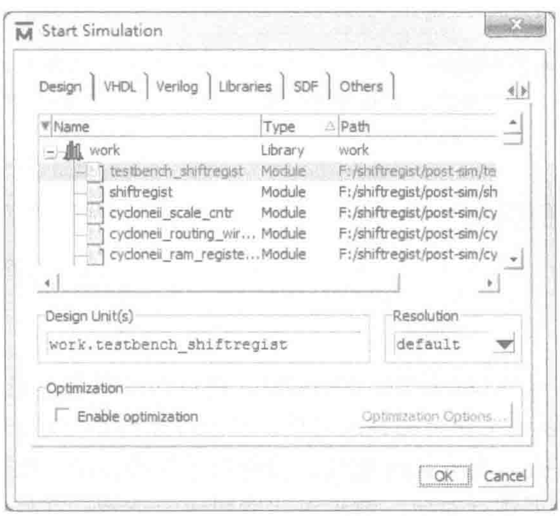


图 7.4-11　选择测试文件

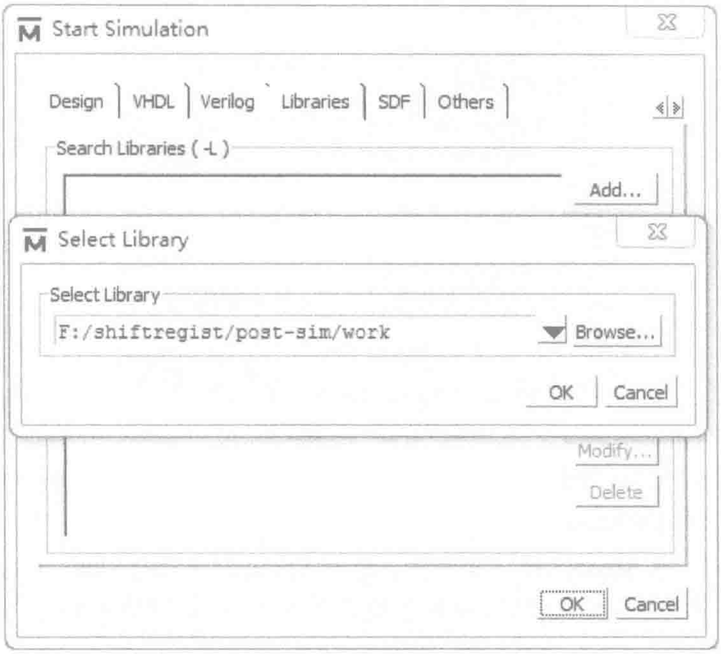


图 7.4-12 添加器件库

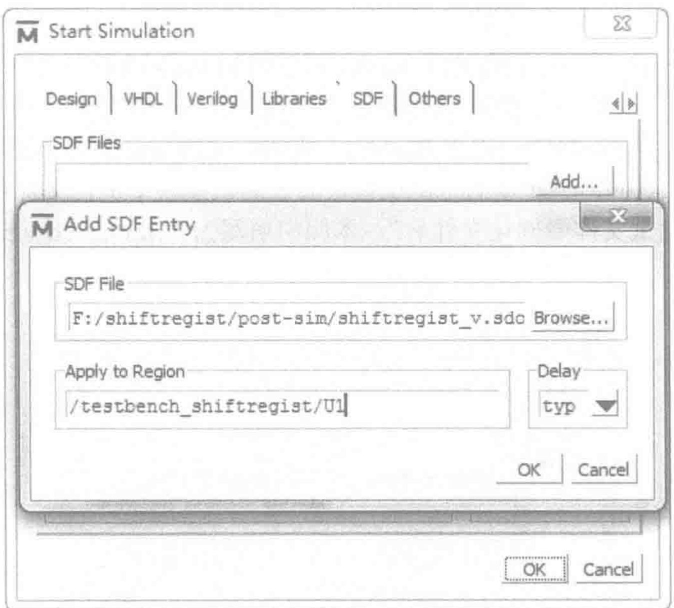


图 7.4-13 添加延时反标注文件

以上工作完成后，点击 OK 按钮，ModelSim 将自动按照设定完成对仿真目标的加载。

(5) 仿真调试。在以上工作全部完成之后，按照 7.2.1 节的方法，运行仿真器。打开波形窗口，将信号添加进去，点击 Run 按钮，得到后仿真的波形，如图 7.4-14 所示。将后仿真的波形与前仿真的波形进行对比。波形对比能快速定位设计在修改前后的区别。在进行波形对比前要先保存原设计的波形文件，该文件称为对比对象。

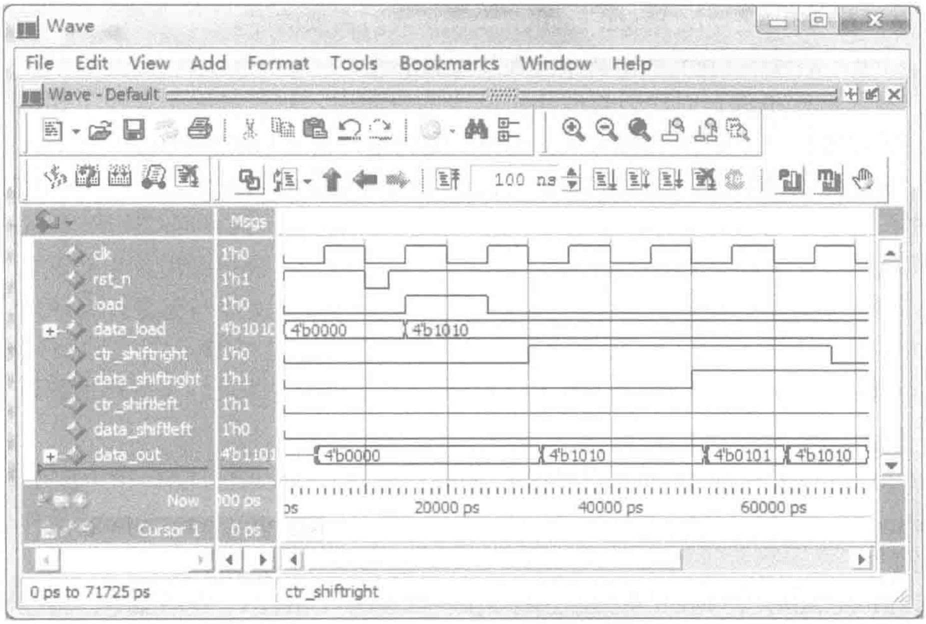


图 7.4-14　后仿真波形

在 ModelSim 中使用波形对比向导，可以方便地完成波形对比功能。具体的工作可以分为如下几个步骤：

① 打开波形对比向导设置。在 Wave 窗口中选择 Tools→WaveformCompare→Comparision Wizard 菜单命令，打开波形对比向导。

② 导入波形文件，作为对比对象。在 Reference Dataset 对话框中，导入原先保存的波形文件，再点击 Next 按钮。

③ 选择对比信号的范围。在四个选项中选择 Specify Comparison by Signnal，对比人为指定的信号，然后点击 Next 按钮。

④ 根据信号范围选择需要对比的信号。在导入波形的所有信号中，选择要对比的信号，然后点击 Next 按钮。

⑤ 分析数据。完成上面几个步骤之后，点击开始比较按钮。ModelSim 软件自动完成波形的对比，并将相关的信息显示在窗口上。

7.4.4　添加仿真库

ModelSim 中的工作库只能有一个，而资源库则可以有许多。其中，在 ModelSim 安装后有些库已经默认安装，例如 IEEE 库中包含预编译的 Synopsys 的 IEEE 算法包，用于仿真加速等功能。这些库都有专门用途，初学者不宜更改。在仿真时，系统可以调用这些资源库来进行仿真，并且这些库是固定不变的。而工作库(默认的为 work 库)只有一个，它用来存放不同设计的编译文件等，并且是不断更新变化的。

ModelSim 仿真中会调用四种常用的仿真库：

(1) 元件库，例如 Stratix 元件库，是在仿真中必用的特定型号的 FPGA/CPLD 的库。

(2) primitive，是调用 Altera 的原语(primitive)设计仿真时需要的库。

(3) Altera_mf，是调用 MagaFunction 设计仿真时需要的库。

(4) Lpm，是调用 Lpm 元件设计仿真时需要的库。

第一种元件库是进行时序仿真时不可缺少的资源库，后三种库是调用了相应的 Altera 设计模块进行设计时才必须用到的库。值得一提的是，在 ModelSim-altera 的 AE 版本中，后三种库是已经编译好的，在 ModelSim-altera 安装目录下的 altera 文件夹中可以找到。

时序仿真需要与具体的器件相对应，这些器件可以自由选择，可以选用 Xilinx 公司的，也可选用 Altera 公司的。ModelSim 并没有自带这些器件库，需要使用者手动添加。在上面测试与综合的实例中，设计者是将用到的器件库和仿真激励文件一起进行编译的，器件库的相关内容直接被编译进了当前的工作库中。下面将介绍在编译代码之前如何永久地在 ModelSim 中添加 Altera 的器件库。这样，使用时只需要调用即可，无需再次编译。

1. 设置工作路径

打开 ModelSim 安装目录(本文中 ModelSim 安装在 C:\Modeltech64_10.4 目录下)，新建文件夹 altera。后面的步骤将在该目录下的 primitive、altera_mf、lpm、stratix 文件夹下存放编译的库。

启动 ModelSim，在 Main 窗口中选择 File→Change Directory 菜单命令，将当前工作路径转到 altera 文件夹；或者在命令行中执行 cd C:\Modeltech64_10.4/altera 命令。

2. 新建库

在 Main 窗口中选择 File→New Library 菜单命令，新建一个名为 primitive 的库。

3. 查找编译资源库所需的文件

在 Quartus Ⅱ安装目录下找到 quartus\eda\sim_lib 文件夹，用于编译资源库的文件有 220model.v、220model.vhd、220pack.vhd、altera_mf.v、altera_mf.vhd、altera_mf_components.vhd、altera_primitives.v、altera_primitives.vhd、altera_primitives_components.vhd、cycloneii_atoms.v、cycloneii_atoms.vhd、cycloneii_components.vhd 等。为方便起见，把它们复制到 altera 中的 Primitive 文件夹下。

4. 编译库

有些书中是把这些文件一起编译的，这适用于 Verilog HDL 和 VHDL 混合仿真，但如果只用一种语言，如 Verilog HDL，则完全没有必要全部编译。这几个文件可分为 Verilog HDL 组(220model.v、altera_mf.v、altera_primitives.v、stratix_atoms.v)和 VHDL 组(220model.vhd、220pack.vhd、altera_mf.vhd、altera_mf_components.vhd、altera_primitives.vhd、altera_ primitives_components.vhd、stratix_atoms.vhd、stratix_components.vhd)。

编译时根据需要编译一组或全部编译。下面以 Verilog HDL 组为例，首先编译 primitive 库。

在主菜单中选 Compile→Compile...命令，查找 altera_primitives.v 文件并对它进行编译，如图 7.4-15 所示。

对 VHDL 文件编译有所不同。LPM 库的 220model.vhd 和 220pack.vhd 可以同时编译；MegaFunction 库中先编译 altera_mf_components.vhd 文件，后编译 altera_mf.vhd 文件；primitive 库中先编译 altera_primitive_components.vhd 文件，后编译 altera_primitive.vhd 文件。元件库，如 Cyclone Ⅱ库中，先编译 cycloneii_atoms.vhd 文件，后编译 cycloneii_components

.vhd 文件；如果是其它系列的元件库，则只要把对应的 cycloneii 改成其它系列的名称即可，例如，若是 cyclone 库，则文件改为 cyclone_atoms.vhd、cyclone_components.vhd 或 cyclone_atoms.v。

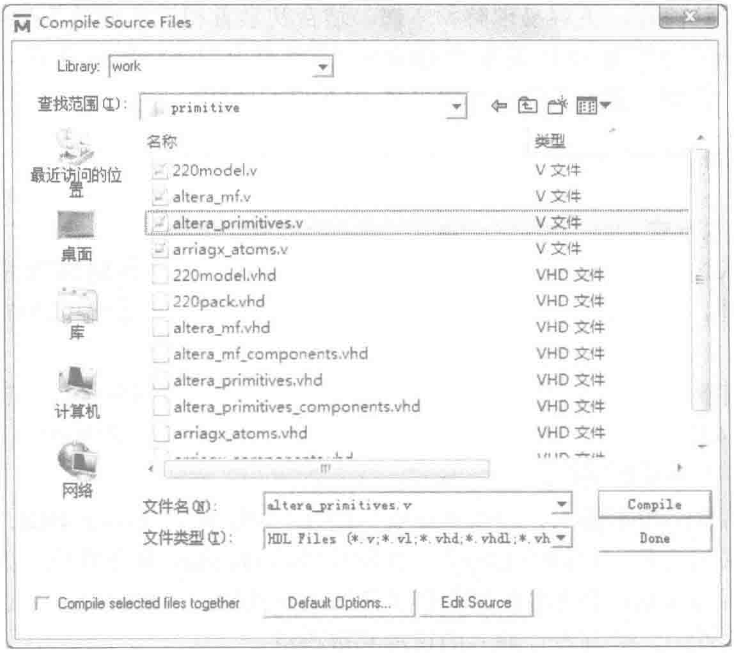


图 7.4-15　对 altera_primitives.v 文件进行编译

5．添加其他的库

重复 2、3、4 步，添加 altera_mf、lpm、stratix 库。

6．配置 modelsim.ini 文件

这一步是为了将编译好的库文件信息添加进系统库，使以后不用再重复添加。

将 ModelSim 根目录下的配置文件 modelsim.ini 的属性(只读)改为可写，用记事本或者其它文本编辑软件打开它。如下所示添加库(注：第一步的工程必须在 ModelSim 的安装目录下才能使用此相对路径)：

```
primitive = $MODEL_TECH/../altera/primitive
altera_mf = $MODEL_TECH/..altera/altera_mf
lpm = $MODEL_TECH/../altera/lpm
stratix = $MODEL_TECH/../altera/stratix
```

注意修改后关闭并改回只读属性。

7.5　Quartus Ⅱ工具全流程应用

Quartus Ⅱ是面向 Altera 公司 FPGA 的集成开发环境，支持原理图、VHDL、Verilog HDL 以及 AHDL 等多种设计输入形式，内嵌逻辑综合器，可以完成从设计输入到硬件配置的完

整的可编程器件设计流程。前几节介绍了使用 ModelSim 进行功能仿真和时序仿真，使用 Synplify Pro 进行逻辑综合，使用 Quartus Ⅱ进行自动布局布线的可编程器件设计流程。本节仍以前述移位寄存器为例，简单介绍一下使用 Quartus Ⅱ进行可编程器件设计的完整流程，相比上述设计流程，更容易理解和掌握，适合初学者和设计要求不高的场合。

本节全流程应用采用 15.0 版本的 Quartus Ⅱ软件并以 Cyclone Ⅴ的 SCSEMA5F31C6 型器件为例进行介绍，要求工作计算机具备 64 位操作系统。

7.5.1　设计实现

1. 设计输入方式

在设计输入过程中可以使用 Quartus Ⅱ框图编辑器、文本编辑器和 EDA 设计输入工具建立用户设计，这些设计中可以包括 Altera 宏功能模块、参数化模块库(LPM)和知识产权核(IP)。

• 框图编辑器：用于以原理图和流程图的形式输入和编辑图形设计信息。Quartus Ⅱ框图编辑器与 MAX+PLUS Ⅱ的 Graphic Editor 类似，能够读取和编辑框图设计文件以及 MAX+PLUS Ⅱ图形设计文件。

• 文本编辑器：用于输入、编辑和查看 AHDL、VHDL、Verilog HDL 等语言文本，以及其它 ASCII 文本文件，包括为 Quartus Ⅱ软件或由 Quartus Ⅱ软件建立的各类文本文件。此外，还可以用 Quartus Ⅱ文本编辑器将 AHDL 语句模板、VHDL 或 Verilog HDL 构造模板插入到当前文件中，提高设计输入的速度和准确度。

2. 设计输入

(1) 首先如 7.4 节所述建立工程，但在添加文件窗口不添加任何已存在的设计文件，而是在 Quartus Ⅱ下新建设计文件，第三方仿真工具选择 ModelSim，第三方综合工具不作选择，默认使用 QuartusⅡ自带的逻辑综合器。然后选择 File→New 菜单命令或单击工具栏的 按钮，弹出如图 7.5-1 所示菜单，选择 Design Files 中的 Verilog HDL File 后，单击 OK 按钮。

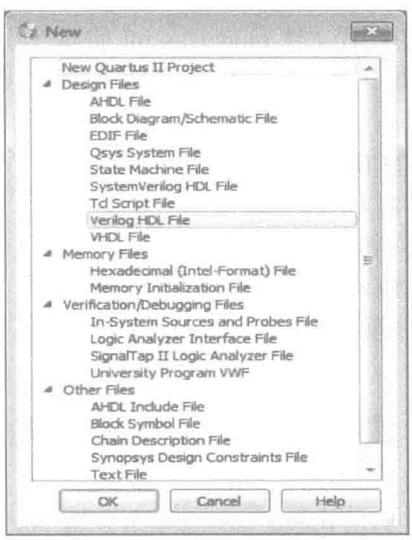


图 7.5-1　新建文件类型选择对话框

(2) 在编辑区写入移位寄存器的 Verilog 代码，并保存至当前工程(注意，该文件是顶层设计文件，建议文件名与实体名一致)。一般保存文件后该文件会自动加入到工程中，若无自动添加，可以选择 Assignments→Settings 菜单命令，在弹出的对话框中选择 Files 项，添加新建立的设计文件。

(3) 执行 Processing→Start→Start Analysis&Elaboration 菜单命令，分析设计，检查设计的语法错误。如果没有报错，则可以进行功能仿真来验证设计的逻辑功能是否正确。

(4) 可以通过选择 Tools→Netlist Viewers→RTL Viewer 菜单命令来显示 RTL Viewer 窗口，其中显示了原理图视图和层次结构列表，还列出了基本单元、引脚和网络，如图 7.5-2 所示。

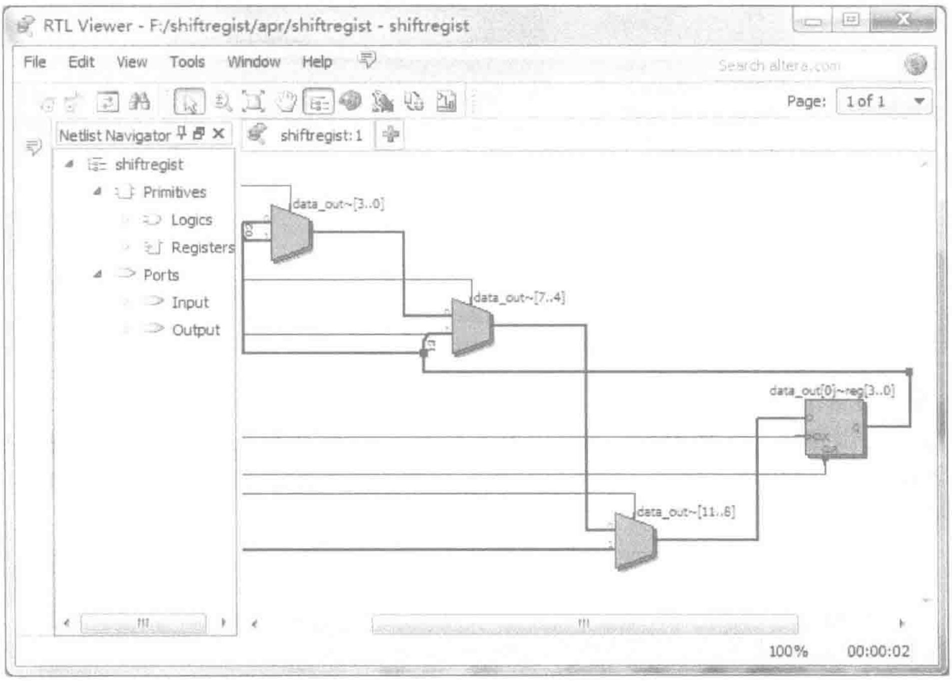


图 7.5-2　RTL Viewer 窗口

如果在层次结构列表中选中某项，则其在相应的原理图视图中高亮显示，反之亦然。RTL Viewer 允许调整视图或者放大、缩小视图来集中查看不同层次的细节，查找特定的名称，并可在层次结构中上、下移动或转到选定的网络来源。

3. 添加设计约束

引脚分配和时序等设计约束通常是设计者编写约束文件并导入到综合、布局布线工具中，在执行逻辑综合、布局布线操作时指导逻辑映射和布局布线。也可以使用 Quartus Ⅱ软件集成的工具 Assignment Editor 和 Settings 对话框等进行引脚分配和时序约束。

(1) 选择 Assignments→Assignments Editor 菜单命令。

(2) 在 Category 栏选择相应的分配类别，它包含了当前器件的所有分配类别，如 I/O Timing(输入/输出时序约束)、Logic Options(区域约束)等。

对于引脚约束，可选择 Assignments→Pin Planner 菜单命令，弹出如图 7.5-3 所示的窗口，

选取某一引脚，根据设计要求在 Location 框选择绑定位置，逐次完成所有引脚的分配。

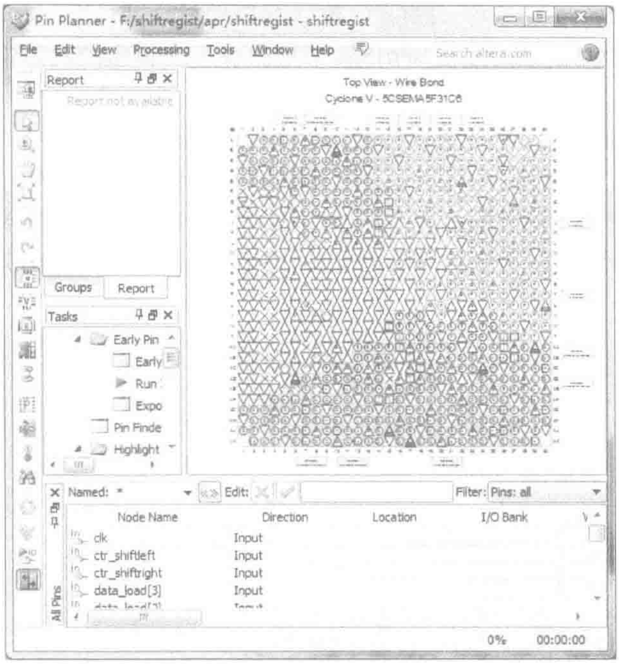


图 7.5-3　引脚锁定界面

4．启动全编译

完成上述步骤后，选择 Processing→Start Compilation 菜单命令或单击工具栏中的 ▶ 按钮进行全编译，如图 7.5-4 所示。编译过程执行完毕后就完成了设计的逻辑综合和自动布局布线。

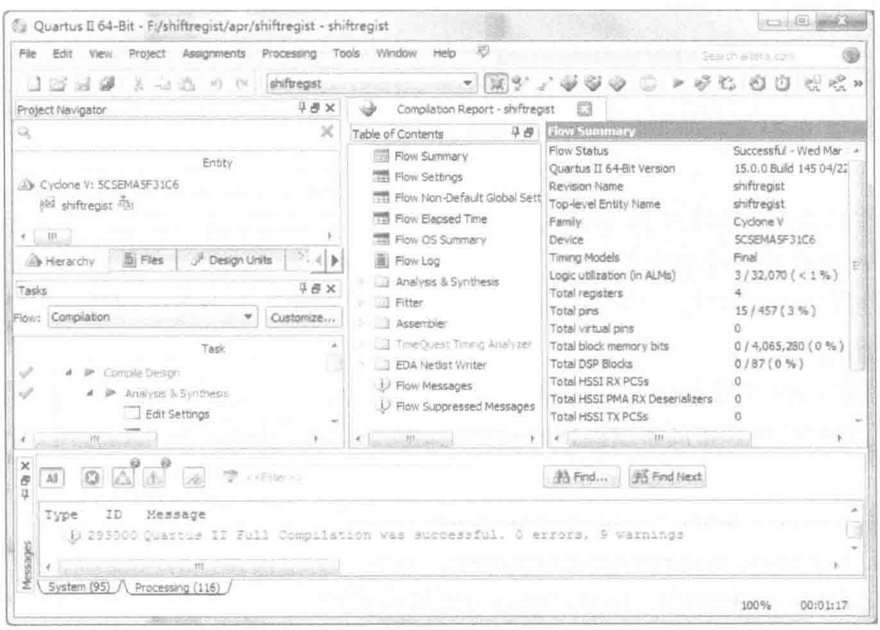


图 7.5-4　编译界面

7.5.2　仿真验证

为了进行仿真，首先需要编写输入测试平台文件。选择 File→New 菜单命令或单击工具栏中的 按钮，弹出如图 7.5-5 所示的对话框。选择 Verilog HDL，单击 OK 按钮，进入代码编辑窗口。



图 7.5-5　选择新建文件类型窗口

(1) 在编辑区编写移位寄存器的测试平台文件，并在测试文件里加上"`timescale 1ns/1ns"，然后保存到当前的工程项目中。测试文件 testbench_shiftregist.v 的 Verilog HDL 程序代码如下：

```
`timescale 1 ns/ 1ns
moduletestbench_shiftregist;
   parametershiftregist_width = 4;
   reg [shiftregist_width-1:0] data_load;
   reg load, clk, rst_n, ctr_shiftright, ctr_shiftleft, data_shiftright, data_shiftleft;
   wire [shiftregist_width-1:0] data_out;
   always   #5 clk = ~clk;
   initial begin data_load = 0; load = 0; rst_n = 1; ctr_shiftright = 0; ctr_shiftleft = 0; clk = 0;
                 data_shiftright = 0; data_shiftleft = 0; end
   initial begin #10 rst_n = 0; #3 rst_n = 1; end
   initial begin #15 load = 1; data_load = 4'b1010; #10 load = 0; end
   initial begin #30 ctr_shiftright = 1; #20 data_shiftright = 1; #17 ctr_shiftright = 0;
                 #20 ctr_shiftleft = 1; #25 data_shiftleft = 1; #20 data_shiftleft = 0; end
   shiftregist U1 (.clk(clk), .rst_n(rst_n), .load(load), .ctr_shiftright(ctr_shiftright),
```

```
            .ctr_shiftleft(ctr_shiftleft), .data_shiftright(data_shiftright),
            .data_shiftleft(data_shiftleft), .data_load(data_load), .data_out(data_out));
endmodule
```

(2) 添加 test bench 文件。选择 Assignments→Settings…菜单命令，调出基本设置窗口，选择 EDA Tool Settings 下的 Simulation 子项。在 Time scale 一栏中设置仿真时间为 1 ns，选择 Compile test bench 选项，单击其后的 Test Benches... 按钮进入下一级设置窗口，如图 7.5-6 所示。

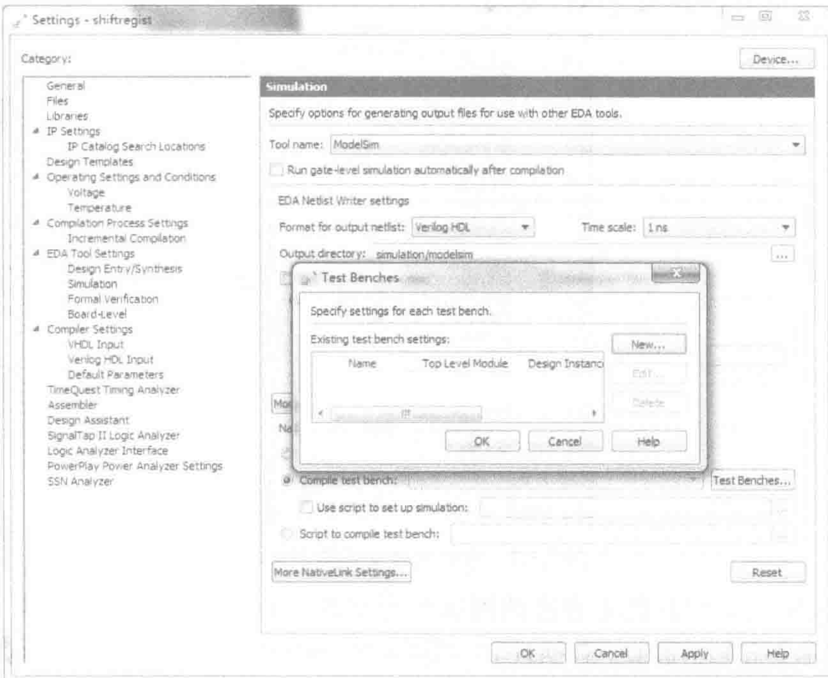


图 7.5-6　新建 Test Bench 窗口

单击 New... 按钮，弹出 Test Bench Settings 窗口，如图 7.5-7 所示。在 Test bench name 一栏中填入 testbench_shiftregist，则 Top level module in test bench 一栏默认设置为 testbench_shiftregist。在 End simulation at 一栏中设置仿真停止时间为 200 ns。最后在 File name 一栏单击其后的 ... 按钮，选择刚才建立的 testbench_shiftregist.v 文件，单击 Add 按钮添加到列表中，最后单击 OK 按钮返回到编辑界面，即完成了 test bench 文件的添加。

(3) 第三方软件调用路径设置。为了能调用 ModelSim 进行仿真，如果 Quartus II 还没有明确第三方软件安装位置则首先需设置其正确的安装路径。选择 Tools→Options…菜单命令，在 EDA Tool Options 子项下对 ModelSim 的安装路径进行设置，如图 7.5-8 所示。

(4) 对整个设计重新进行全编译，然后选择 Tools→Run Simulation Tool→RTL Simulation 菜单命令或单击工具栏中的按钮，Quartus II 自动启动 ModelSim 软件进行功能仿真。仿真结果如图 7.5-9 所示。

仔细检查时序仿真波形是否符合预定逻辑功能。如果不符合则返回修改 Verilog 代码，再次进行仿真，直到仿真结果符合设计预定功能为止。



图 7.5-7　Test Bench 设置

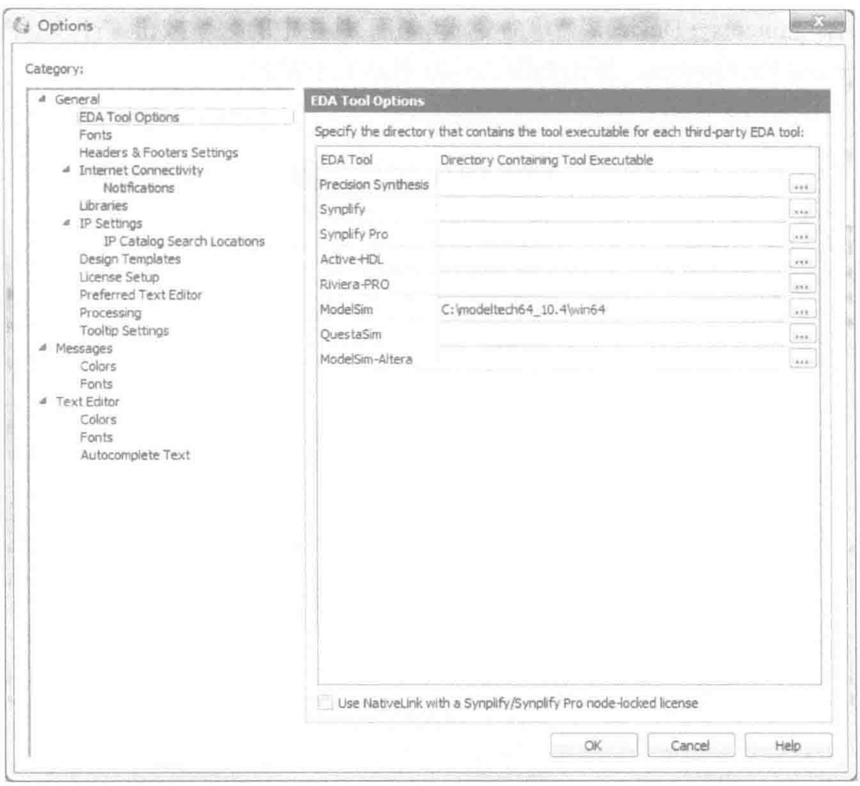


图 7.5-8　ModelSim 仿真软件路径设置

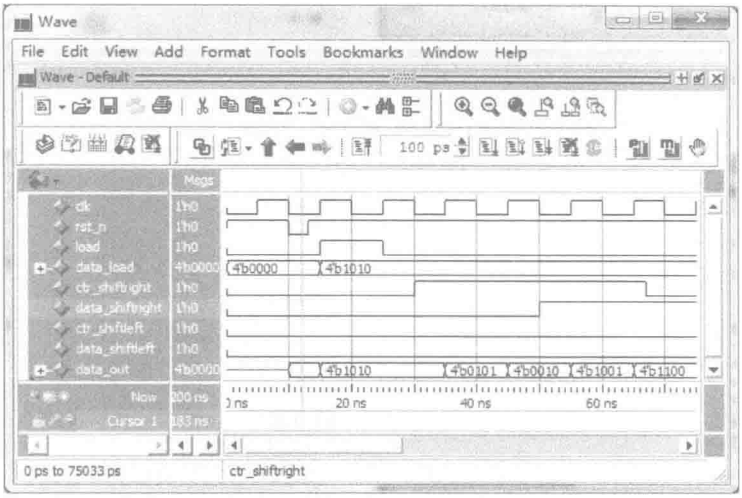


图 7.5-9　功能仿真结果

关于时序仿真，由于 Quartus II 15.0 并不支持 Cyclone V 器件的门级时序仿真(Gate-level timing simulation)，但是可以通过 TimeQuest 静态时序分析而不是通过适配后时序仿真来查看时序信息。具体操作请参阅相关专业文档。

7.5.3　编程与配置

选择 Assignments→Device...菜单命令或单击工具栏中的按钮，打开器件设置窗口，选择 Device and Pin Options，弹出如图 7.5-10 所示的子窗口。

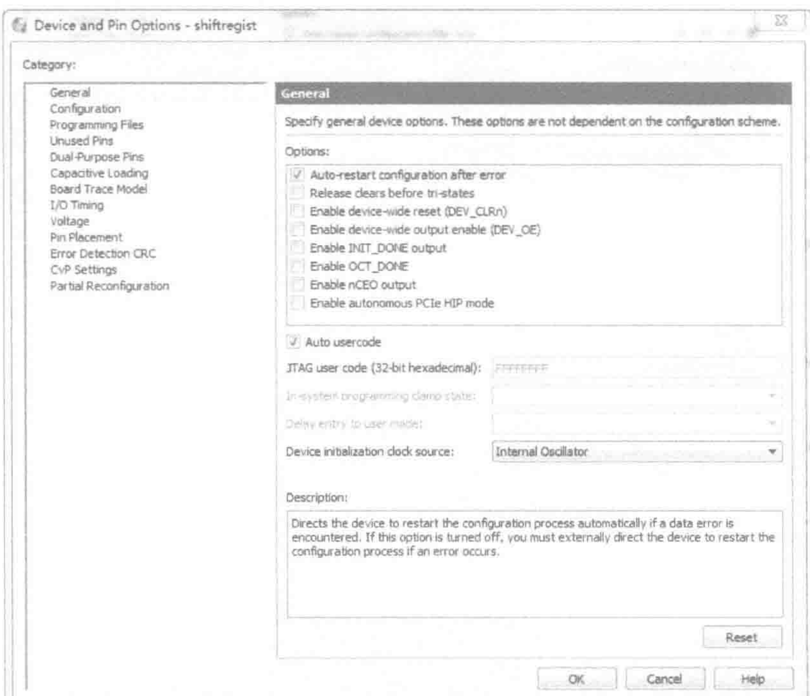


图 7.5-10　Device and Pin Options 子窗口

在 General 选项卡下的 Options 栏中选中 Auto-restart configuration after error(通常该选项默认选择)，切换到 Programming Files 选项卡，如图 7.5-11 所示，设置生成文件类型。

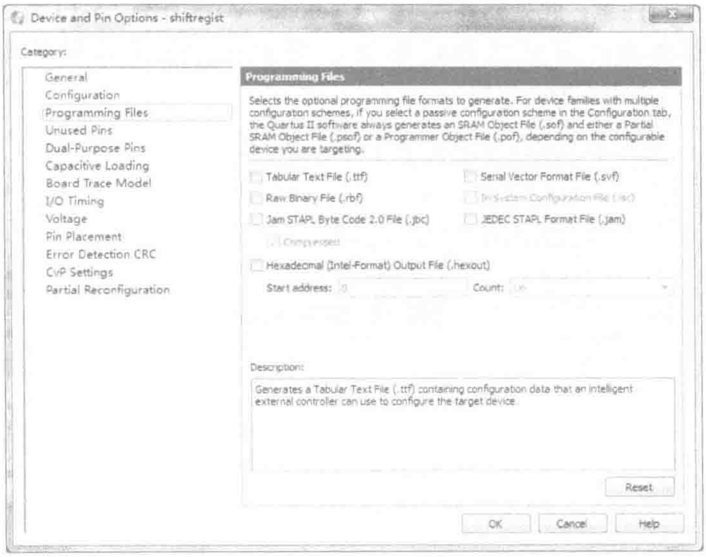


图 7.5-11　可选辅助编程文件选项卡

允许指定可选编程文件格式，例如十六进制(Intel 格式)输出文件(.hexout)、表格文本文件(.ttf)、原二进制文件(.rbf)、Jam 文件(.jam)、Jam 字节代码文件(.jbc)、串行向量格式文件(.svf)和在系统配置文件(.isc)，设计者可根据需要进行勾选，这里不作设置。

下面以 JTAG 编程模式为例，介绍编程文件的配置过程。

选择 Quartus Ⅱ窗口的 Tools→Programmer 菜单命令或点击图标，进入器件编程和配置界面，如图 7.5-12 所示。

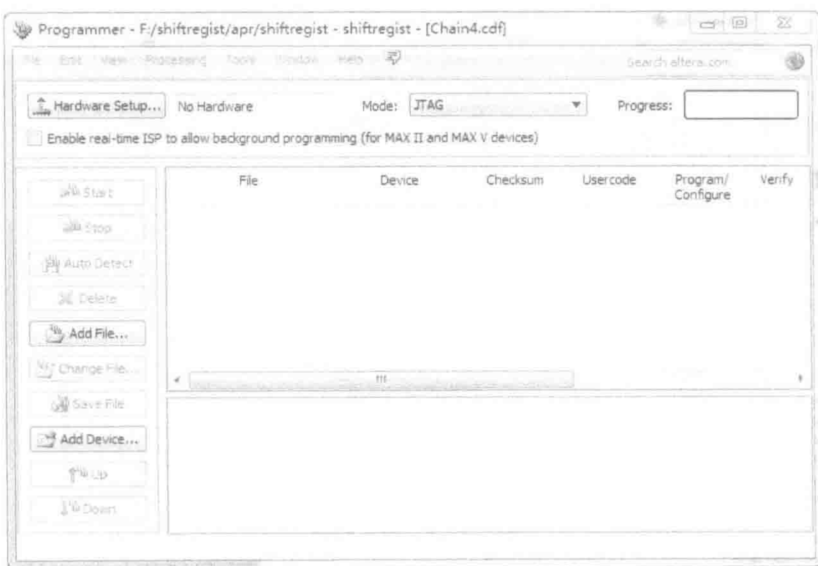


图 7.5-12　器件编程和配置界面

如果 Hardware Setup...后为 No Hardware，则需要选择待编程的硬件。点击 Hardware Setup...，进入 Hardware Setup 对话框，点击 Currently selected hardware 栏，若出现 DE-SoC[USB-1]选项(注意，此处器件类型为 Cyclone Ⅴ)，如图 7.5-13 所示，则表明已安装好实验箱仿真器驱动；若不出现此选项，则需要安装实验箱仿真器驱动。选择 DE-SoC[USB-1]，再点击 Close 按钮。

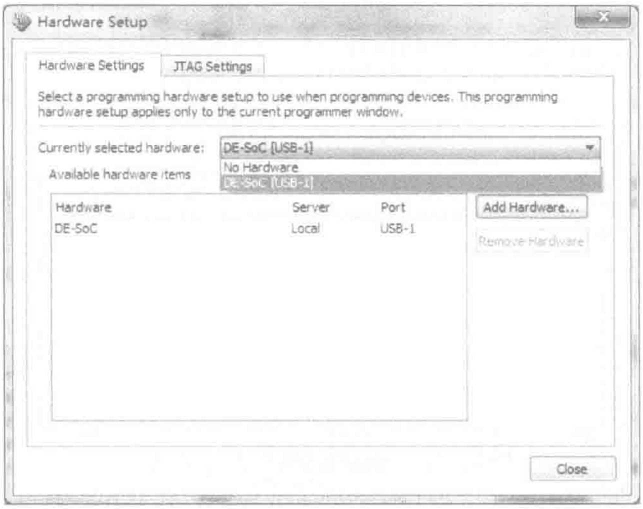


图 7.5-13　硬件选择框

在器件编程和配置界面，选择下载模式，在 Mode 中指定的编程模式为 JTAG 模式。确定编程模式后，单击 Add Device... 按钮添加器件，如图 7.5-14 所示，在 Device family 下选择 Cyclone Ⅴ，在 Device name 下选择 5CSEMA5F31，再点击 OK 按钮。

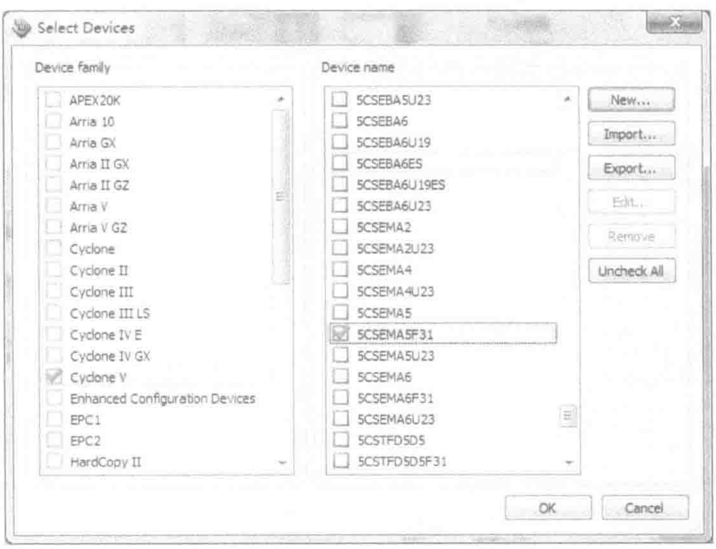


图 7.5-14　器件选择

单击 Add File... 按钮，在 F:\shiftregist\apr\output_files 路径下添加相应的 shiftregist.sof 编程文件，勾选 shiftregist.sof 文件后的 Program/Configure 选项，然后点击 Start 按钮

下载设计文件到器件中，Progress 进度条中显示编程进度，如图 7.5-15 所示。编程下载完成后就可以进行目标芯片的硬件验证了。

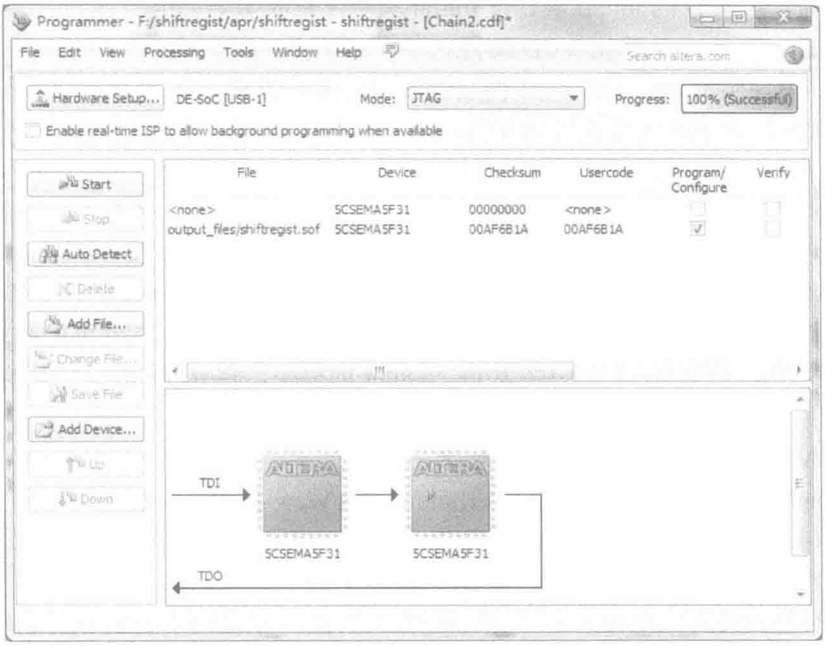


图 7.5-15　编程完成

本 章 小 结

本章介绍了典型的数字集成电路设计流程，给出了每个阶段的主要任务和目前常用的软件；常用仿真工具 ModelSim 和 NCSim 的基本使用流程及调试手段；Synplify Pro 和 Design Compiler 的基本使用；利用一个带左移右移控制的移位寄存器，结合 ModeSim、Synplify Pro 和 Quartus Ⅱ软件介绍了仿真、综合、布局布线全过程，并且简要说明了如何将常用的库文件信息添加进 ModelSim；介绍了只使用 Quartus Ⅱ进行可编程设计的完整流程。

思考题和习题

1. 结合数字集成电路的设计流程图，思考每个阶段的主要任务。
2. 为什么在 0.18 μm 以下的设计时采用物理综合？
3. 什么是建立时间和保持时间？
4. 比较静态时序分析和动态时序分析的优缺点。
5. 功能仿真和时序仿真有何区别？
6. 波形窗口、数据流窗口和列表窗口这三种窗口在 ModelSim 仿真调试过程中分别有怎样的特点？如何正确使用这三个窗口进行仿真调试？

7．针对给出的移位寄存器的例子，重新编写 Testbench 测试向量，验证其功能是否正确。

8．编写控制 VCD 的 initial 块。该 initial 块必须完成如下工作：

(1) 设置 mfile.dmp 为输出 VCD 文件。

(2) 转储模块实例 top.a1.b1.c1 中两层深度内的所有变量。

(3) 在时间单位 200 处停止转储 VCD。

(4) 在时间单位 400 处开始转储 VCD。

(5) 在时间单位 500 处停止转储 VCD。

(6) 创建一个检测点，把所有 VCD 变量的当前值转储到 VCD 文件。

9．在 ModelSim 中创建一个 DO 文件，完成以下任务：新建 work 库，将 work 库映射到当前的工作目录，编译 counter.v 和 tb_counter.v 文件(这两个代码文件可以在 ModelSim 的 Example 中找到)，仿真 work 库中名为 tb_counter 的模块，将 tb_counter 中所有的信号加进波形图中。

10．在 ModelSim 中如何进行波形对比？

11．Synplify 综合后的 RTL 视图和 Hierarchical View 视图有何区别？

12．如何从综合报告中分析最差路径信息以及时序信息？

13．综合过程中各种库文件的含义和区别是什么？

14．一个全减器有三个输入 x、y 和 z(前面的借位)，以及两个输出 D(差)和 B(借位)。D 和 B 的逻辑等式如下：

$$D = x'y'z + x'yz' + xy'z' + xyz$$

$$B = x'y + x'z + yz$$

要求：为全减器编写 Verilog HDL RTL 描述；使用身边的工艺库综合该全减器；优化电路，使其达到最快速度；把同样的激励应用于 RTL 和门级网表上，比较它们的输出。

15．使用 Verilog HDL RTL 描述设计一个 3 线-8 线译码器。给译码器提供三位输入 a[2:0]，译码器的输出是 out[7:0]，由 a[2:0]索引到的输出位的值是 1，其它位是 0。要求：使用身边现有的工艺库综合该译码器；优化电路，使其面积最小；把同样的激励应用到 RTL 和门级网表，比较它们的输出。

16．什么是延时反标注文件，如何生成延时反标注文件？

17．自己编写一段产生正弦波形的代码，用 ModelSim 分别做前仿真和后仿真，用模拟的方式查看波形以及加入时延信息后波形的变化。

18．ModelSim 仿真时工作库、资源库的概念是什么？

19．利用 7.4.4 节的方法将自己常用的库文件信息编译添加进 ModelSim 的系统库中。

第 8 章　设计与验证语言的发展趋势

以 Verilog HDL 和 VHDL 为代表的硬件描述语言(HDL)的发展至今已有 30 多年的历史，并成功地应用于设计的各个阶段：建模、仿真、验证和综合等。20 世纪 80 年代出现了上百种硬件描述语言，对设计自动化曾起到了极大的促进和推动作用。在这些硬件描述语言中，Verilog HDL 和 VHDL 最终成为业界的标准化语言。

近年来，微电子技术飞速发展，在制造工艺上从 350 nm 提高到 14 nm，为功能更强、更复杂的芯片制造提供了基础，数字集成电路的规模和复杂度也迅速提高。在传统硅工艺上，集成电路正从单一的数字或模拟芯片向数模混合的单芯片结构发展；芯片结构正从高频率单一内核向低频率异构多内核体系发展；单芯片集成晶体管数量也从几万门增加到了几十亿门。这些新技术和新需求的产生为 ULSI 芯片设计方法和验证方法提出了新的要求，归结起来主要有两方面。

(1) 如何提高 ULSI 规模的 ASIC 和 FPGA 的设计效率。大规模、高复杂度、数模混合是数字集成电路的发展趋势，应用在多媒体、通信系统、计算设备等领域的异构 SoC 和 MPSoCs 结构芯片已经成为集成电路新的增长点。如何在现有的硬件描述语言和方法学上，针对新的设计问题，提出有效的解决方法和途径是目前集成电路 EDA 领域需要解决的最主要问题。

Verilog HDL 和 VHDL 等硬件描述语言在解决中小规模数字集成电路的设计上发挥了强有力的作用，然而集成度和规模的扩大使得传统硬件描述语言在设计效率上产生了新的问题。

应用在通信系统和多媒体领域的集成电路需要进行大量的复杂计算模型仿真和定点化设计，如图 8-1 所示，然后用硬件描述语言设计电路，如果不符合设计要求，就要修改数学模型，直到达到设计要求，再送到下一个阶段进行芯片设计。

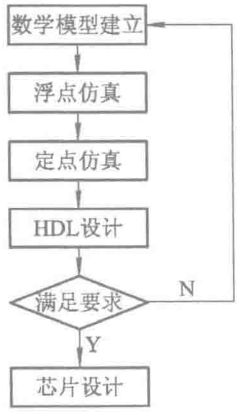


图 8-1　SoC 设计流程

Verilog HDL 和 VHDL 在这种系统级设计方面显得力不从心，而 C、C++ 和 Matlab 等程序设计语言在系统级设计和仿真中显示出了强大的设计效率。这就导致了目前 ULSI 芯片设计的尴尬局面，即系统级设计采用高级程序设计语言，芯片设计和仿真采用灵活性较差的硬件描述语言。这种局面使得数字集成电路的设计人员和部门分工变得复杂，严重影响了设计效率。设计人员希望能有一种高效的设计语言将系统设计和电路设计结合在一起，以面对复杂的大规模集成电路的设计。

(2) 如何极大地提高电路规模和复杂结构芯片设计、仿真的效率和有效性。仿真和测试是确保数字集成电路芯片流片成功的关键。在目前的工艺水平和成本下集成电路的流片费用高，已经成为芯片成本的重要部分。如何有效通过仿真和测试确保芯片流片的成功率，是硬件描述设计语言面对的一个问题。

当面对中小规模集成电路时，Verilog HDL 和 VHDL 可以有效地从电路分析的角度进行硬件设计的测试和仿真。然而目前集成电路规模巨大，功能丰富，模拟和数字结构混合，给测试和仿真带来了巨大困难。同时，系统设计往往采用 C 语言等高级程序设计语言。这些情况一方面使得采用传统硬件描述语言的测试和仿真时间长，复杂程度高；另一方面使得系统验证和电路验证采用不同平台，一致性和可靠性降低。由于硬件电路测试和仿真时间长、复杂度高，同时也是确保芯片质量的重要环节，因此高效率的测试和仿真方法是硬件描述语言面对的另一个问题。

在 2001 年举行的国际 HDL 会议上，与会者就使用何种设计语言展开了激烈的辩论。最后就“如果要启动一个芯片设计项目，愿意选择哪种方案”进行了投票表决，结果，仅有极少数票赞成使用 SystemC 为代表的高级程序设计语言，而有约 40 票赞成采用以 Verilog HDL 为代表的传统硬件描述语言。

Verilog HDL 支持者认为，开发一种新的设计语言是一种浪费，Verilog HDL 有能力面对超大规模数字集成电路在设计方法方面的新困难；SystemC 支持者认为，系统级芯片 SoC 快速增长的复杂性需要新的设计方法；C 语言的赞扬者认为，Verilog HDL 是硬件设计的汇编语言，而编程的标准很快就会是高级语言，CynlibC++ 是最佳的选择，它的速度快、代码精简；Superlog 的捍卫者认为，Superlog 是 Verilog HDL 的扩展，可以在整个设计流程中仅提供一种语言和一个仿真器，与现有的方法兼容，是一种进化而不是一场革命。还有人提出设计带有模拟电路的芯片，硬件描述语言必须有模拟扩展部分，如 Verilog HDL-A，既要求能够描述门级、开关级，又要求具有描述物理特性的能力。

目前的情况是，以 Verilog HDL 和 VHDL 为代表的传统设计语言依旧是数字集成电路的最主要设计语言，以 System Verilog 为代表的传统硬件描述语言逐渐在测试仿真中开始使用，以 SystemC 和 C 语言为代表的高级程序设计语言还在应用和推广的初期阶段。对于 ULSI 规模的集成电路设计和验证，面对的问题依然存在，设计语言还在不断的探索和完善中。

下面对目前比较有希望的硬件描述语言作一个简单介绍。

1. System Verilog

随着高级验证语言(如 OpenVera 语言)的出现，Co-Design Automation Inc 发明了 Superlog 语言。随后，Co-Design Automation Inc 被 Synopsys 收购，Superlog 和 Vera 的基础被捐献给 Accellera。之后，Superlog 转换升级为 System Verilog。

Accellera 在 2002 年 6 月推出了下一代 Verilog 标准 System Verilog，System Verilog 几乎可以弥补现有 Verilog 的所有缺陷。System Verilog 结合了来自 Verilog HDL、VHDL、C++的概念，还有验证平台语言和断言语言，也就是说，它将硬件描述语言(HDL)与现代的高层级验证语言(HVL)结合了起来，使其对于进行当今高度复杂的设计验证的验证工程师具有相当大的吸引力。这些都使得 System Verilog 在一个更高的抽象层次上提高了设计建模的能力。它主要定位在芯片的实现和验证流程上，并为系统级的设计流程提供了强大的连接能力。

System Verilog 集成了 Verilog HDL 的简洁、C 语言的强大、功能验证和系统级结构设计等特征，是一种高速的硬件描述语言，其体系结构如图 8-2 所示。

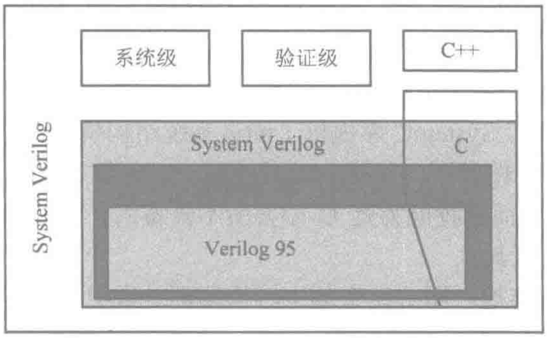


图 8-2　System Verilog 体系结构

System Verilog 是 Verilog HDL 的超集，支持最新的 Verilog HDL 95 和 Verilog HDL 2001 的硬件模型。Sysytem Verilog 提供 C 语言的结构、类型、指针，同时具有 C++面向对象的特性和扩展综合子集 ESS(提供一种新的硬件描述的综合抽象级)，并且具有强大的验证功能和自动测试基准，如随机数据产生、功能覆盖、各种专有检查等。

System Verilog 3.0 标准于 2002 年 6 月发布，主要扩展是系统级设计和 IP 设计的建模。这些扩展为使用 C/C++、SystemC 的高层次系统体系结构工程师和 RTL 级、门级实现的硬件工程师提供了沟通的桥梁，使两组人员可以在同一种语言上进行通信而无需语言转换。

System Verilog 3.1 标准于 2003 年 5 月发布，主要扩展是先进的验证和 C 语言整合能力。

2005 年 11 月，IEEE 通过了由 Verilog HDL 新标准和 System Verilog 统一成的标准语言 IEEE 1800-2005 System Verilog——统一的设计、说明和验证语言。

IEEE 1800-2005 System Verilog 扩展了 IEEE 1364 Verilog，可用于体系结构级模型的建立和验证，标准还包括设计说明方法、嵌入断言语言、具有覆盖和断言 API 的测试程序(Testbench)语言，以及直接编程接口(DPI)。新标准能提高设计和确认的效率，包括设计、模拟、确认、基于断言的形式验证流等的效率。

2．SystemC

SystemC 系统级描述和验证语言基于 C++，是为熟悉 C\C++的人而设计，是一种软硬件协同设计语言。它具有较高的抽象能力，同时能体现出硬件设计中的信号同步、时间延迟、状态转换等物理信息，可以给工程师提供一个系统级设计的公共基础平台。在常用的设计语言中，C、C++和 Java 等高级编程语言有较高的抽象能力，但由于不能体现硬件设计

的物理特性，硬件模块部分需重新用硬件描述语言设计，使得后续设计缺乏连贯性。

SystemC 是在 C++ 的基础上扩展了硬件类和仿真核形成的，由于结合了面向对象编程和硬件建模机制原理两方面的优点，使得 SystemC 可以在抽象层次的不同级进行系统设计。系统硬件部分可以用 SystemC 来描述，其基本单元是模块(Model)，模块内可包含子模块、端口和过程，模块之间通过端口和信号进行连接和通信。

随着通信系统复杂性的不断增加，工程师将更多地面对使用单一的语言来描述复杂的 IP 和系统，而 SystemC 语言具有良好的软硬件协同设计能力。

SystemC 是一个 C++ 库，也是一种使设计者可以有效地设计出一个软件算法的准确循环模型、硬件结构以及进行系统级设计的方法。设计者可以用 SystemC 开发工具或在标准 C++ 开发工具中添加如 SystemC 库等来制作系统级模型，快速地仿真和优化设计，以及研究不同的算法，并且为硬件和软件设计人员提供一个设计系统的可执行规范。可执行规范本质上是一个 C++ 程序，它显示了和设计系统同样的性能，为软件设计人员和硬件设计人员提供了一个设计的标准。SystemC 库提供了创造系统结构模型的必需结构，包括了那些在 C++ 中没有的功能，如硬件时序、并行和触发功能。C++ 这种面向对象语言提供了通过增加类来扩展语言的能力，而这种能力是 C 语言所不具备的，因此，SystemC 使用的是大家熟悉的 C++ 语言和开发工具。

SystemC 基于 C++ 结构，图 8-3 中的上层构架都是很明确地建立在下层的基础上的。SystemC 内核提供一个用于系统体系结构、并行、通信和同步时钟描述的模块；完全支持内核描述以外的数据类型、用户定义数据类型；通常的通信方式，如信号、FIFO，都可以在内核的基础上建立，经常使用的计算模块也可以在内核基础上建立。

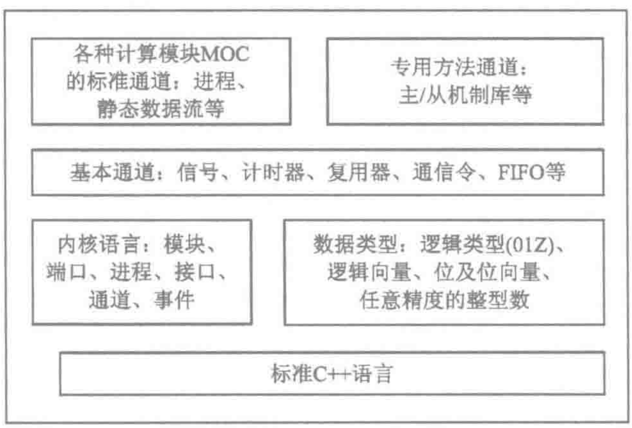


图 8-3　SystemC 体系结构

在实际使用中，SystemC 由一组描述类库和一个包含仿真核的库组成。在用户的描述程序中，必须包括相应的类库，可以通过通常的 ANSI C++编译器编译该程序。SystemC 提供了软件、硬件和系统模块。用户可以在不同的层次上自由选择，建立自己的系统模型，进行仿真、优化、验证、综合等等。

SystemC 是由 Synopsys 公司和 CoWare 公司合作开发的。1999 年 9 月 27 日，40 多家世界著名的 EDA 公司、IP 公司、半导体公司和嵌入式软件公司宣布成立“开放式 SystemC 联盟”。著名的 Cadence 公司也于 2001 年加入了 SystemC 联盟。SystemC 从 1999 年 9 月

联盟建立初期的 0.9 版本开始更新，从 1.0 版到 1.1 版，直到 2001 年 10 月推出了最新的 2.0 版。

本 章 小 结

本章以 Verilog HDL 和 VHDL 两种硬件描述语言为代表，分析了集成电路设计过程中设计与验证语言的发展趋势，阐述了目前形势下对 ULSI 芯片设计方法和验证方法提出的新要求，最后对目前前景较好的硬件描述语言 System Verilog 和 SystemC 作了一个简单介绍。

思考题和习题

1. 通过对硬件描述语言发展历程的了解，简要回答如今集成电路设计所面临的挑战和问题。
2. 简要回答 System Verilog 和 SystemC 各自的特点。

参考文献

[1] 袁俊泉，孙敏琪，曹瑞. Verilog HDL 数字系统设计及其应用. 西安：西安电子科技大学出版社，2002.

[2] Samir Palnitkar. Verilog HDL 数字设计与综合. 夏宇闻，等译. 北京：电子工业出版社，2004.

[3] 王金明. Verilog HDL 程序设计教程. 北京：人民邮电出版社，2004.

[4] 云创工作室. Verilog HDL 程序设计与实践. 北京：人民邮电出版社，2009.

[5] Michael D Ciletti. Verilog HDL 高级数字设计. 张雅绮，等译. 北京：电子工业出版社，2005.

[6] 常晓明，李媛媛. Verilog HDL 工程实践入门. 北京：北京航空航天大学出版社，2005.

[7] Bhasker J. Verilog HDL 入门. 夏宇闻，甘伟，译. 北京：北京航空航天大学出版社，2008.

[8] 张明. Verilog HDL 实用教程. 成都：电子科技大学出版社，1999.

[9] 刘福奇，刘波. Verilog HDL 应用程序设计实例精讲. 北京：电子工业出版社，2009.

[10] 罗杰. Verilog HDL 与数字 ASIC 设计基础. 武汉：华中科技大学出版社，2008.

[11] 吴戈. Verilog HDL 与数字系统设计简明教程. 北京：人民邮电出版社，2009.

[12] Bhasker J. Verilog HDL 综合实用教程. 孙海平，等译. 北京：清华大学出版社，2004.

[13] 王冠，黄熙，王鹰. Verilog HDL 与数字电路设计. 北京：机械工业出版社，2006.

[14] 张延伟，等. Verilog HDL 程序设计实例详解. 北京：人民邮电出版社，2008.

[15] 夏宇闻. Verilog HDL 数字系统设计教程. 北京：北京航空航天大学出版社，2008.